EXHIBIT D

LookUp Controller (LUC) High Level Design

Revision 1.29

April 6, 2002

Simon Knee

Revision History

Revision	Date	Author	Description of Changes
0.10	June 7, 2001	Simon Knee	Initial document. Released to Vitek.
0.11	June 13, 2001	Simon Knee	Version given to Vitek, Robert, Fazil.
0.12	June 15, 2001	Simon Knee	Pre-review version.
0.13	July 24, 2001	Simon Knee	Version given to Nitesh for Configuration HLD.
0.14	August 1, 2001	Simon Knee	Version given to Oki.
0.15	August 13, 2001	Simon Knee	Version given to Vitek.
0.16	August 28, 2001	Simon Knee	Workspace format changed to 116-bit flow key. All other parts of document still need to be changed for 116-bit flow key. Modifications made with post-review comments. Still more modifications to be made, after which this can become a v1.00 document.
0.60	November 1, 2001	Simon Knee	Extensive modifications for B10/S10, TACL, split workspaces etc. Correct Configuration Register section to match that of the RTL. This version is now feature complete. Changes will be tracked more accurately from this version forward.
1.20	November 19, 2001	Simon Knee	Post review comments and bug fixes: • The DUMP_IDL and FEED_IDL flags of the STATUS register are not used during initialisation. • PR 67 Fix: Figure 35: Process Delete Entry Flow Chart and Figure 36: Process Release (RELS) Command modified to use a new FDC_ERR bit of the STATUS register. • PR 59 Fix: Figure 40: Process Expired Timer Subroutine and section 3.6.6 modified to include the Socket ID in a Timer Event to the Dispatcher. • PR 69 Fix: Section 3.6.1 on TCU variables and Table 24: TCU M10CNT and M8CNT Counters incorrectly stated which timer tables entries have the 2s tick. • PR 73 Fix: Collapsed the eight TD_ERROR bits of the STATUS register into a single bit. • Added some text to describe Figure 36: Process Release (RELS) Command. • PR 78 Fix: Table 33: Workspace Response Values was missing a *Valid Processor Core Response* for the TACL case. • PR 77 Fix: Reduced the workspace write listen masks (see section 4.8.6.1) to 16-bits since maximum listen size is 1KB. Modified Figure 23: Write Workspace Subroutine so that it uses these new registers, sets upper 16-bits of mask to zero for listen write back. • PR 80 Fix: Figure 27: Process Listen Lookup incorrectly said, "Command is LFKC and TACL enabled". It should have been LFLKC. • Clarified that the LFKC command only sends a workspace header when a new flow is created. See section 2.4.3. • PR 81 Fix: Figure 3: TCP Flow Key Format had the *Receive Interface* as Flow Key[15:0], should be Flow Key[15:8]. Modified the section on TACL (section 2.5) to include the default allow/deny bit of the CONFIG register. Table 5: TACL Entry Field Descriptions: values of *Listen Fixer Pointer* were incorrect. • PR 97 Fix: Figure 27: Process Listen Lookup and the associated text were fixed so that on a LLKC command we perform the TACL lookup after the listen lookup. • The explanations of the INIT and FILLI bits of the CONTROL register were not correct. Correct that description, and fixed the initialisation sequence to match. • Completed section 3.8 on *Expected Performance*.

Revision	Date	Author	Description of Changes
			• Added a description of the LUC Timer Queue (see section 4.2.2). Added the LTQ_FULL bit to the STATUS register, to indicate if the LUC Timer Queue ever became full. • Updated Figure 14: High Level LUC Block Diagram to more closely match the RTL. • Updated the section on *TACL Table Management* to more clearly describe the sequences that should be followed for adding / deleting entries. • Added the EN_RBG bit to the CONTROL register. This enables sampling of the Random Bit Generator. See section 4.8.5. • Cleared up issues to do with initialisation of the Available Tables. See section 2.7. • ~~Made it clear that during a teardown the Processor Cores need write~~ no workspaces back. • Made it clear that it is valid if neither the Protocol Core nor the Interface Core writes back a workspace. • Renamed the *Global Timestamp* to *Global Timer*. See section 2.10. • Specified that if an out of range Core ID is presented to the LUC in a LUC command then the OPC_ERR bit of the STATUS register is set (see section 4.8.4). • Figure 33: ULID Flow Chart was updated to show that the workspace write back and workspace validity clear are not done in parallel. • Figure 23: Write Workspace Subroutine had multiple errors to do with address calculation, now corrected. • Figure 27: Process Listen Lookup and Figure 30: Process Listen Create modified so that the TACL lookup is done after the hash list length is checked.
1.21	December 4, 2001	Simon Knee	Bug fixes: • PR 125 Fix: Added the RELS_CNT register (see section 4.8.62). • Made it clear that the ADDR_ERR bit of the STATUS register only detects invalid MMC addresses, and not DDR addresses. • PR 129 Fix: Figure 30: Process Listen Create was not using the correct response for the TACL denied case. • PR 128 Fix: Figure 35: Process Delete Entry Flow Chart would increment TEARDOWN_CNT even if the FDC gave an error. It should not have done. Figure 36: Process Release (RELS) Command had the same issue, now fixed. • PR 141 Fix: Figure 23: Write Workspace Subroutine, *Write Back Interface Core Area* box had incorrect *to* equation for the DDR destination. It used ddr_socket_base + pc_len / 4 + i, when it should have been just ddr_socket_base + i. • PR 142 Fix: Bit fields of LIS_WR_MASK register were incorrect (see section 4.8.61). • PR 182 Fix: DDR Refresh Cycle (DDR_RCYC) of the TCU_CONTROL register (see section 4.8.10) so that "DDR_RCYC * 4 * T_{CLK} is less than or equal to 7.8us." • Table 8: LUC Parameters, clarified that if one core uses an area as read-only then it may be possible to do a Protocol / Interface core split on a 128-bit boundary. • Made it clear in sections 2.3.5.4 and 3.4.2 that the first chunk of a workspace **must** be written back when a flow is created. • Clarified section 3.5 to say that we are only restricted on the number of LUC engines for the *same type of operation (lookup or update)*.

Revision	Date	Author	Description of Changes
1.22	December 21, 2001	Simon Knee	Bug fixes: • Table 16: Global Timer Signals had a typo: IMER instead of TIMER. • Updated Figure 1: LUC Overview to remove the *ND Queue* of the Dispatcher. No LUC changes required for this. • PR 252 Fix: Modified the EN_RECV bit of the CONTROL register so that is only enables the Dispatcher Interface: the Message Bus interface can no longer be held off since no backpressure is available.
1.23	January 15, 2002	Simon Knee	Bug fixes: • PR 306 Fix: Modified *Figure 30: Process Listen Create* so that ENTRY_NCNF_CNT is incremented on a TACL deny. • PR 314 Fix: *Receive Interface* was only 11-bits in *Figure 11: B10 TACL Entry Format*, should have been 12-bits. A description of the *Receive Interface* fields was also added to Table 5: TACL Entry Field Descriptions. • PR 325 Fix: STLINE_CNT field added to the CONTROL register (see section 4.8.6). • Typo fixes from Anil.
1.24	January 24, 2002	Simon Knee	Bug fixes: • PR 384 Fix: Figure 25: Lookup Command Flow Chart and Figure 34: Teardown Commands (TDFK, TDLK) Flow Chart modified so that the listen hash is computed using the flow key and listen mask. • PR 391 Fix: Modified section 4.8.1.2 to say that after each write the WRITE_BUSY bit must be polled. • PR 398 Fix: Correct initialisation sequence in section 4.9. The ACT, SBINIT and SRESET bits of the CONTROL register are now used correctly. • Default value given to the STLINE_CNT field of the CONFIG register (section 4.8.6).
1.25	February 19, 2002	Simon Knee	Bug fixes: • PR 625 Fix: Updated sections 2.3.5.3 and 2.3.5.4 on flow state splitting so that 64-byte boundaries are now a requirement. Also updated various register definitions to make this requirement more explicit. • PR 642 Fix: Updated section 4.8.27 to include a better description of what registers contain what DDR words. • PR 644 Fix: Some register names updated to match Hardware Reference Manual.
1.26	March 13, 2002	Simon Knee	Bug fixes: • PR 657 Fix: Section 4.8.58 modified to say that (FLW_SH_SIZE + 2) must be a multiple of 64-bytes, rather than FLW_SH_SIZE. Same for LIS_SH_SIZE. • PR 754 Fix: Updated the section on the Flow Locker (3.2) so that it includes the special case scenarios. HL_DENY_LIM parameter added to the HASH_PARAMS register (4.8.23). • PR 780 Fix: *Table 34: Timer ID* to *Timer Mapping* updated to show correct timer mapping. • PR 792 Fix: Section 2.11 on *Error Correction and Detection* and updated the DEBUG register (section 4.8.26) to allow for forcing of ECC errors. • PR 837 Fix: Added section 4.8.1.3 explaining which register values are dynamic, and which are static. • PR 838 Fix: Sections 2.10.2 and 4.8.5 updated so that they use the correct GLOBAL_TIMER signal, and indicate that the GTM_CLR bit must be set to value 1 and then value 0.

Revision	Date	Author	Description of Changes
			• PR 874 Fix: Section 2.3.5.6 on the rules of writing back workspaces was added. Also fixed up a number of sections that said it is ok for no workspaces to be written back on a USID (this is not valid). • PR 895 Fix: Renamed OPC_ERR to GEN_ERR in the STATUS register (section 4.8.4). Noted that it now checks for USID with no workspace write back. • PR 898 Fix: Modified the TCU_CTRL register (section 4.8.10) so that the INTERVAL field is wider, shifted the DDR_RCYC value left by four.
1.27	March 19, 2002	Simon Knee	Bug fixes: • PR 950 Fix: Added the *Trace Sequence Number* field to *Figure 42: Workspace Format* and *Table 32: Workspace Field Descriptions.*
1.28	April 3, 2002	Simon Knee	Bug fixes: • PR 1077 Fix: Added the DDR_R2WTURN bit to the LUC_PARAMS register (section 4.8.9).
1.29	April 6, 2002	Simon Knee	Bug fixes: • PR 1077 Fix: Fix made in version 1.28 was not 100% correct. If DDR_R2WTURN is turned on then an extra 2 clocks are added. Section 4.8.9 was updated.

Table of Contents

List of Figures

List of Tables

1 Introduction

In this document we describe the high level operations of the LookUp Controller (LUC). This document should describe all the necessary behaviour of the LUC, without enforcing any particular implementation method.

Throughout this introduction (section 1), we provide an insight into the functionality of the LUC. This is provided as introduction material only, and for full details the reader should examine the complete document.

1.1 Related Documents

Document	Revision	Author
Astute Content Processor Architecture Presentation	N/A	Fazil Osman
Dispatcher High Level Design	1.29	Simon Knee
Event Specification	1.20	Simon Knee
Flow Director CAM (FDC) High Level Design	1.30	Simon Knee
Host Message API (Level 3) High Level Design	1.23	Billy Oostra
Input Processing Unit (IPU) High Level Design	1.36	Bob Sefton
ManageMent Controller High Level Design	1.10	Nitesh Mehta
Message Bus High Level Design	1.30	Mark Feinstein
Output Processing Unit (OPU) High Level Design	1.23	Bob Sefton
Packet Processor High Level Design	1.19	Octera
Protocol Cluster High Level Design	1.14	Kirk Larson
Queuing Model Trace of a Simple HTTP Request	1.00	Simon Knee
Socket Memory Controller High Level Design	1.10	Simon Knee
Some Statistical Plots of Web Traffic	1.00	Brian Petry
Scratchpad High Level Design	1.48	Charles Kaseff
SPI-4 Module High Level Design	1.02	Bob Sefton
TCP Processing Paths for the Content Processor	1.00	Simon Knee, Brian Petry
TCP/IP Algorithm Review	0.91	Brian Petry
TCP/IP Core Software: Functional Specification and Conformance Statement	1.10	Brian Petry
TCP/IP Core Software: High Level Design	WIP	Brian Petry

1.2 Overview

During the operation of protocols such as TCP, we must often look up the state of a particular connection. For TCP this state is keyed by the {IP Destination Address, IP Source Address, TCP Destination Port, TCP Source Port} tuple. The amount, and content, of this state varies widely between protocols. For example, a flow state for TCP would be 512 bytes long and would contain information such as:

- Connection state (Established, Closing, Closed etc.)
- Window sizes.
- Timer information.
- Route information.
- Pointers to data buffers for receive and transmit.

Similarly, once the TCP protocol has been processed there may be some form of application processing required. This application processing also requires state information. The amount of application state required obviously depends on the application being supported.

The purpose of the LUC is to store this protocol and application state information in a format that allows for quick lookups and updates. The LUC calls this information *flow state*, and it accesses it via a flow key. For TCP this flow key consists of the {IP Destination Address, IP Source Address, TCP Destination Port, TCP

Source Port, IP Protocol, Receive Interface} tuple, but for other protocols other fields will be used. The receive interface is included in the flow key for when we wish to distinguish flows received on different interfaces. For traditional TCP termination this receive interface is set to zero by the Packet Processor. It is mainly used for transparent TCP termination.

Depending on the context, this document will sometimes use the term flow key, and will sometimes spell out the individual fields of the flow key.

Another task of the LUC is to maintain timers on a per flow basis. Protocols and applications often require such timers for e.g. frame re-transmission, delaying packets etc. With large numbers of flows maintaining these timers can be a large overhead for a general purpose CPU, therefore requiring that the LUC take control of this task.

Figure 1 illustrates the position of the LUC with regards to the other functional units. The LUC interfaces to the FDC using the LUC_FDC_Bus and FDC_LUC_Bus via the Dispatcher[1]. The LUC uses the FDC as an atomic unit to co-ordinate its behaviour with the Dispatcher. The LUC also interfaces to the Dispatcher via the Dispatcher_LUC_Bus for LUC commands from the Dispatcher to the LUC, and via the LUC_Dispatcher_Bus for timer events from the LUC to the Dispatcher. The LUC also has an interface to an external random number generator: it uses this to pass random number seed information to the protocol cores. The LUC has an interface to the MMC bus for management and control.

Finally, the LUC has an interface to the 266MHz DDR memory that it uses to store flow state. For the S10 part this is a maximum of 4GB of memory that is dedicated to the LUC for Flow State. It is accessed via a 128-bit wide bus.

For the B10 part the LUC shares a maximum of 16GB of DDR memory with the SMC. In this case the LUC can use up to the first 4GB of DDR memory for Flow State. It is accessed via a 64-bit wide bus.

For efficiency, the minimum unit of transfer in the DDR memory is 64-bytes for the S10 and 32-bytes for the B10.

[1] The Dispatcher passes these commands through to the FDC without any alteration, and echoes the response back to the LUC.

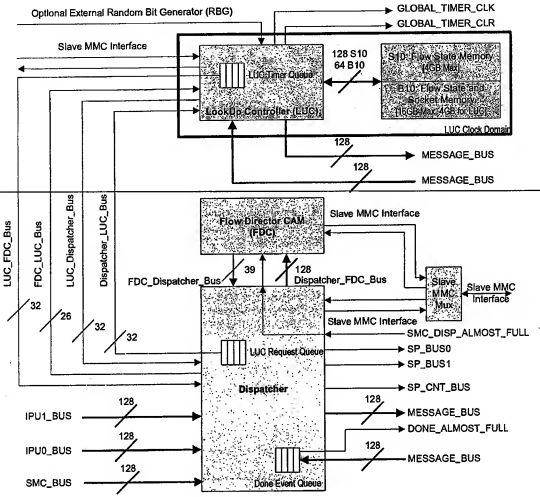

Figure 1: LUC Overview

1.2.1 TCP Termination

In this document we use the term *TCP Termination* to refer to the case where the ACP acts as the end point of a TCP connection, i.e. it runs the TCP state machine, buffers data, re-assembles the byte stream, and does all of the duties of a regular TCP client. There are two modes that a TCP Termination device can operate in: Non-transparent and Transparent.

1.2.1.1 Non-Transparent Termination

In this scenario the remote peer of the TCP connection is communicating with us via a virtual IP address. The ACP then transports the byte stream of that TCP connection to the application that is being hosted. If the application host CPU is local, then a direct connection between the ACP and host CPU is used, and no other network protocols are involved. However, if the application host CPU is on another server, i.e. if we are acting as a proxy, then a new TCP connection must be initiated to that server, and the data copied between one connection and the other.

When that new connection is initiated with the server, for non-transparent termination we use a new source IP address and new TCP source port number. Since we use new addressing information, we are not transparent to the server, and we have hidden the address of the true client.

1.2.1.2 Transparent Termination

Transparent termination only applies for proxy connections. The difference between a transparent and non-transparent connection is that for transparent connections we re-use the client addressing information in the connection to the server, i.e. rather than use our own IP address and TCP source port, we use the same IP address and TCP source port that the client supplied. We are effectively transparent to the server since it cannot tell we are in the middle of the byte-stream.

An interesting point about transparent termination is that the same {IP Destination Address, IP Source Address, TCP Destination Port, TCP Source Port, IP Protocol} tuple can appear on two different interfaces but reference two different TCP connections. One TCP connection would be from the client to the ACP, the other from the ACP to the server. To distinguish these two TCP connections we must introduce an extra field into the tuple: the receive interface. For transparent termination the receive interface should be set to a value that distinguishes the client connection from the server, e.g. perhaps we use the 802.1Q VLAN ID or simply an external port number.

For non-transparent TCP termination the receive interface should be set to the same value for all receive interfaces. In fact, this is an IP host requirement since the same TCP flow is allowed to arrive on any IP interface.

1.2.2 TCP Packet-by-Packet Forwarding

The ACP also supports TCP Packet-by-Packet Forwarding. This mode of operation only really applies after a TCP proxy has been created. Note that the TCP proxy itself should be created by the correct termination of a TCP session (to get the URL) and negotiation of a new connection with the server, i.e. even TCP Packet-by-Packet Forwarding requires some element of TCP Termination to start the connection.

Once a proxy has been established it is possible to simply translate packets on one connection to packets on another. Re-writing the address information, and applying a TCP sequence number delta allows us to do this. In this mode we are not running the TCP state machine, we are not buffering any significant amounts of data, and we are not re-assembling the byte stream. The TCP state machines are instead running on the client and server, and we are merely translating packets.

This mode of operation normally requires less state to be held per flow entry, and can sometimes provide faster processing.

1.2.3 How Many Flows?

An obvious question is how many flows should the LUC be able to simultaneously track? The answer to this question depends on the type of application being supported. For example, for layer 7 switches there will be large numbers of clients, therefore requiring that the LUC support a large number of flows. For IP storage based applications there is likely to be a small number of flows, perhaps in the thousands.

Since we must support both these types of applications the LUC must be configurable in how many flows are supported. The LUC will support anything from 64K flows to 4M. Using more or less flows should not significantly impact performance, but it will allow the external memory size to be flexible.

When quoting performance metrics we assume that the LUC is in a mode that supports up to 1M flows, i.e. the performance metrics are not valid if the LUC has been configured for more than 1M flows.

1.2.4 How Large is each Flow?

Now that we have fixed on a range for the number of flows, how much state do we need to store per flow? This depends on both the application that is being supported, and the protocol that the application is using.

The reason is that not only do we need to store protocol state, but we also need to store session information that the application can use, e.g. application state may include such things as "how many bytes left to the next SSL record header or iSCSI protocol data unit".

The current requirements state that TCP requires approximately 448 bytes for state information. If we then assume that an application requires 64 bytes of session information, then that gives the flow state size at around 512 bytes.

When quoting performance metrics we assume that the flow size is 512 bytes, i.e. the performance metrics are not valid if the LUC has been configured for a flow size larger than 512 bytes.

To be flexible, the LUC must allow for flow sizes ranging from 128 bytes to 2048 bytes.

1.2.5 Split Protocol Core and Interface Core Processing

The ACP is designed such that two processor cores can be assigned to process a flow: a protocol core and an interface core. The protocol core might take care of TCP related functions, while the interface core would perform more application-oriented tasks. Both of these processor cores will require the LUC to store state regarding a flow.

The LUC manages this task by splitting a workspace into two parts: one part is for the protocol core, and one part for the interface core. This split is configurable, and could even be such that one core gets all of the flow state and the other gets none[2]. The split also allows a region of the flow state to be marked as shared, i.e. it is sent to both the Protocol Core and the Interface Core.

When the LUC is given a request to perform a lookup it is informed of the protocol core ID, the protocol workspace ID, the interface core ID and the interface workspace ID. Once the LUC has found the flow state, it then sends the appropriate amount to each {Core ID, Workspace ID} pair. After the processor cores have finished processing, they will write back their respective workspaces to the LUC.

The processor cores then negotiate with the Dispatcher to indicate that they have finished processing the event. When the Dispatcher has determined that both cores have finished processing, it sends an update command to the LUC. The LUC then re-assembles this back into a single workspace and updates the flow.

1.2.6 Automatic Flow Creation

During the normal operation of the LUC there will be times when it cannot find any flow state information for a given flow key. This would be the normal situation when a protocol is creating a new connection. Given the performance requirements of the Astute Content Processor, it is not reasonable to expect that a general purpose CPU can create these flows. Instead we must allow the LUC to manage its own data structures and create a flow entry. If the LUC creates a flow that the general purpose CPU would not have done[3] then the CPU has the ability to tear it down. To reduce this effect, flow creation may be further qualified by the contents of a frame. For example, if the protocol is TCP then the LUC can be programmed such that flows are only created for frames that have the SYN flag set and only the SYN flag[4].

1.2.6.1 Listen Lookups

Certain protocols allow a network stack to offer specific services. For example, a TCP running on a server may be offering a web service, but not pop-3 or telnet. When a TCP connection is started to the server, the destination port indicates what type of service is being requested: web is port 80, pop-3 is port 110 and telnet

[2] In this case the LUC would not even send out the second workspace.
[3] Flow creation must follow the rules of the protocol we are implementing. Since the LUC does not understand these protocols it may create a flow when the protocol would not have.
[4] In reality it is a Packet Processor that examines the frame contents and decides if it should allow the LUC to create a flow. The interface core would also be able to create flows using a special event type: this is an active TCP open.

is port 23. If a connection is opened to a port on a server, and that server does not support that service (port), then the server simply replies with a "connection refused" message. This method of determining if a server is "listening" on a given service (port) is called a listen lookup.

The LUC allows any fields of the flow key to be used in a listen lookup[5]. For a typical TCP application the listen lookup may consist of checking if the {IP Destination Address, Destination Port, IP Protocol} tuple exists in a listen database. The IP Destination is included since a server may have multiple IP addresses, and for each IP address we may support multiple services.

Flow creation can now be even more finely tuned by using a listen lookup, i.e. if the flow lookup does not find anything then a listen lookup is performed. If a listen entry is found then we must be interested in that service, so a flow entry is created[6]. On the next lookup request for this flow we will find a match in the flow table. If a listen entry is not found then a flow is not created.

1.2.6.2 Termination Access Control Lists

Some applications require an even stricter method for determining when to create a flow. In this scenario the application would like to validate the flow key against a set of rules, to see if this flow is allowed to have an entry created for it. This is rather like a firewall operating at layers 3 and 4. The rules that the flow key is checked against are commonly referred to as access control lists (ACL). Since this check is only performed when we attempt to create a flow entry (start a termination), we call them Termination Access Control Lists (TACLs).

The LUC has the capability to perform a TACL lookup after a flow key lookup does not find an entry[7]. This TACL lookup will either allow or deny the creation of a flow entry. Note that we do not perform a TACL lookup if we find a flow entry: we assume that it has previously been validated.

The LUC can support the following types of conditions in a TACL:

1. An exact match on the IP Protocol
2. Mask and match on the IP source address, i.e. the incoming source address is first masked and then compared against a value.
3. Mask and match on the IP destination address.
4. Range match on the TCP/UDP source port. This is a {from, to} range.
5. Range match on the TCP/UDP destination port.
6. Range match on the Receive Interface.

The LUC compares an incoming flow key against each rule in the TACL in turn, starting with the first rule. It will stop at the first match it finds, or when it runs out of rules. The result of an ACL lookup is either to allow or deny a listen lookup / flow creation.

Table 1 illustrates an example TACL that the LUC can support. In this example the first rule allows anyone on the 172.21.2.* network[8] to contact port 80. Note how the IP Destination is effectively excluded from the match by setting its mask to 0.0.0.0. Similarly, the source port is effectively excluded from the match by putting in the full range of source ports as the {from, to} value. The second rule also allows a specific source address range, but this time a destination address is given, and a range of port numbers is used. The last rule in the table is a catchall rule. This rule will match with any flow key, and effectively denies any flow keys

[5] This is a global setting that applies to all flows.
[6] In some cases a flow entry is not created: we simply report that a listen entry was found.
[7] TACL can also be applied to listen key lookups where a listen key will be created if one is not found.
[8] Rather than 172.21.2.*, the term 172.21.2 / 24 can be used, where the 24 is the number of significant bits in the network address.

that did not match either the first or the second rule. Note that the LUC keeps a default allow / deny value that is used if no rules match, so this final rule is not required[9].

IP Prot.	IP SA	SA Mask	IP DA	DA Mask	Src. Port Start	Src. Port End	Dest. Port Start	Dest. Port End	Rcv. I/F Start	Rcv. I/F End	Allow or Deny
TCP	172.21.2.0	255.255.255.0	0.0.0.0	0.0.0.0	0	65535	80	80	0	4095	ALLOW
TCP	198.1.128.0	255.255.128.0	10.3.2.1	255.255.255.255	0	65535	80	100	0	4095	ALLOW
TCP	0.0.0.0	0.0.0.0	0.0.0.0	0.0.0.0	0	65535	0	65535	0	4095	DENY

Table 1: Example Termination Access Control List (TACL)

Since the access control lists are only checked when a flow is being created, these are not strictly speaking the same access control lists that many routers implement. For example, for stateless events[10], there is never any ACL checking performed. This is the reason why we call this termination access control lists (TACL).

It should be noted that the performance metrics quoted in this document assume that TACL lookups are not enabled. If TACL lookups are enabled then the connection per second performance will be reduced, perhaps significantly if a large number of TACL rules are present.

1.2.6.3 Listen Fixers

As described in section 1.2.6.1, listen lookups can be configured to include certain parts of the flow key. For example, the LUC could be configured to use {IP DA, TCP Destination Port} as the listen key. However, this configuration is global and applies to all flows. For example, it would not be possible to create some listen entries that look at just the {IP DA, TCP Destination Port} and other listen entries that look at {IP DA, TCP Destination Port, IP SA}. This function is present in a BSD like TCP/IP stack, where you can restrict the source IP addresses that can connect to a listening socket.

Consider the following example. Let us suppose that the listen entry contains information about how many buffers should be allocated to flows that use this service etc. We may then want to create two listen entries: one for a small number of class A customers, and another for everyone else, who by default is class B. With our current method for listen entries this would require including the IP SA in all listen lookups, and then enumerating each and every customer's IP address. That could require thousands of listen entries.

Note that if we turn on TACL then we can recognise the class A customers with a small number of rules using the IP SA. The problem is how do we carry this match information onto the listen lookup? This is done using the concept of a *listen fixer*. A listen fixer is a 16-bit field that is stored for each TACL entry. If a match is made against a TACL rule then this listen fixer can be written into any 16-bit aligned area of the listen key. How would this be used in our example? Well suppose that there were a 16-bit area of the listen key that is normally masked out, i.e. is normally zero. If a TACL rule matches a class A customer then we could set the listen fixer to a value N, so that the listen key gets modified. We then need only two listen entries: one with the value N in the listen fixer area (class A), and the other with zeros in those bit positions (class B).

1.2.7 Flow Coherency

When a frame arrives at the ACP the LUC finds the associated flow state and passes this information to a protocol and interface core. The processor cores then process the frame, most likely updating the state of the flow. This state is then written back to the flow state memory of the LUC. Since there are multiple protocol and interface core pairs executing simultaneously, we must ensure that the flow state is correctly updated and interpreted.

[9] In fact, using a rule in the TACL table would consume extra DDR bandwidth for flow keys that did not match.

[10] Stateless events do not cause a LUC command to be issued: they have no state associated with them. An example is the event caused by the reception of an ARP packet.

One solution to this problem is to have the protocol and interface cores lock a flow state while they are working on it. However, this introduces considerable delay since a processor core cannot start working on a flow until all other processor cores unlock it. This also requires a reasonable amount of hardware to perform the locking on all of these flows.

The approach that has been taken for the ACP is to ensure that while there are active events for a flow, all processing for that flow is done on the same protocol and interface core pair. Once all currently active events have been processed, the protocol and interface core pair are un-assigned from that flow. On the next set of events we may then pick a different pair of processor cores. This has the effect that during the lifetime of a flow we may perform processing on a variety of protocol and interface core pairs.

Given that a single protocol and interface core pair is processing a flow, there is no need to do a lookup for every single event: on the first event a lookup is performed and the flow state is written to the workspaces of the protocol and interface cores. If future events arrive before that event has been processed, we simply pass the event onto the same protocol and interface core pair and use the version of the workspace that they already have. In fact, the processor cores have probably modified the state of the flow due to the first event, so it is a requirement that the LUC does not update this flow state with the version it has in its own memory. It should be noted that this requirement for not performing lookups on subsequent events is not a requirement for the LUC: other functional units perform this task and simply do not submit lookup requests to the LUC.

It should be noted that this method of assigning protocol and interface core pairs does mean that for a given flow we only have the compute power of a single protocol core and single interface core, i.e. maximum performance from the ACP is achieved with multiple flows.

The device that ensures that the same protocol core is used for events on the same flow is the Flow Director CAM (FDC). The flow key forms the key of this CAM, and returns the protocol core and interface core that have been assigned, along with their respective workspace IDs. The Dispatcher uses the FDC to direct events to the correct protocol and interface core pair. For more information on coherency the reader is directed towards the FDC and Dispatcher HLD documents.

1.2.8 Timers

Each flow entry in the LUC has a small number of associated timers. These timers are used to correctly execute the protocol, e.g. re-transmission timeouts etc. The reason the LUC performs this task is that with 4 million flows maintaining these multiple timers is quite a burden for a general purpose CPU.

When the LUC has determined that a timer has expired, it creates a timer entry in the FDC and passes a "timer expired" event to the Dispatcher. Eventually the Dispatcher will service this timer expiration, and will request a flow lookup. A Processor Core pair is then notified of the timer expiration and they use the flow's state to determine the correct action and next state. An important aspect here is that the LUC does not automatically send the flow state to the Processor Cores: it must wait until the FDC indicates that it can do so. The reason is that timer events have a critical section with normal packet or interface events, as explained below.

During the processing of a packet event the protocol core may decide to reset a timer. To do this it will set a *reset* bit for that timer, and when the LUC is told to write this state back to flow memory it will correctly adjust its timers. Note however that there is a time gap between when a protocol core decides to cancel a timer, and when the LUC actually cancels the timer. Given this fact, consider a case where a protocol core is processing a packet event and it decides to cancel the re-transmission timer. However, just at that instant in time the LUC expires the re-transmissions timer, and issues a timer expiration event to the same protocol core: this is not correct since the timer should have been cancelled. The protocol core will now receive a re-transmission timeout, which violates the TCP protocol.

To prevent this timer critical section, two methods are employed by the LUC:

1. The LUC is not allowed to issue timer events for flows that are *checked out*. A flow is checked out from the time the LUC receives a lookup request for that flow, to the time the LUC receives an update / teardown for that flow. However, due to queues between the LUC and the Dispatcher, this requirement does not cover 100% of the critical section, so we require another rule:
2. The LUC must issue a create timer command to the FDC before a timer expired message can be sent to the Dispatcher. If the FDC has an entry for the flow then the create timer command will be denied, and the LUC must back off and try again later.

These details are discussed in more depth in future sections.

2 Functional Operation

2.1 Number Schemes

In the following sections we describe the numbering schemes used for the Core ID, Core Bitmaps and Workspace ID. Note that the schemes for Core ID and Core Bitmaps are identical to those for the FDC and Dispatcher.

2.1.1 Core ID

We use the term Core ID to refer to a number that describes either a protocol core or an interface core in the *system*.

Cores are allocated to a cluster. There are three clusters in total, with each cluster containing five cores. Figure 2 illustrates the format that is used to create a Core ID. The Core ID is basically constructed from two bits of Cluster Number, followed three bits of Core Number. This Core ID format is the format used on the Message Bus. Note that for three clusters with five cores per cluster, the valid Core ID values are 0, 1, 2, 3, 4, 8, 9, 10, 11, 12, 16, 17, 18, 19, 20. Specifically, the Core ID values 5, 6, 7, 13, 14 and 15 are not valid.

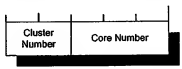

Figure 2: Core ID Format

2.1.2 Core Index

A Core Index is an encoding of the Core ID, as shown in Table 2. A Core Index provides a number in the range 0 through 14 with no holes, unlike a Core ID that has holes at 5, 6, 7, 13, 14, and 15.

Core Index	Core ID
0	0
1	1
2	2
3	3
4	4
5	8
6	9
7	10
8	11
9	12
10	16
11	17

Core Index	Core ID
12	18
13	19
14	20

Table 2: Core Index to Core ID Mapping

2.1.3 Core Bitmaps

In some circumstances we need to keep a bitmap of cores. The question is in this bitmap what Core ID does bit i represent? One obvious choice is that bit i represents Core ID i. The problem with this is that not all Core ID's are valid, so the bitmap would be larger than it really needs to be. Since bitmap width is important, another encoding is used to store Core Bitmaps. Instead we use a bitmap such that bit i represents Core Index i. Table 2 can then be used to convert this Core Index into a Core ID.

2.1.4 Workspace ID

We use the term Workspace ID to refer to an address in a Processor Cores workspace area of DMEM. The Processor Cores workspace DMEM is split into sixteen equal sized chunks, and each chunk is given an incrementing Workspace ID starting at zero. For example, if the workspace area of DMEM were 2KB then each Workspace ID would address to 128 bytes. If each Workspace ID addressed 128 bytes, and each Workspace was size 512 bytes, then only Workspace IDs 0, 4, 8 and 12 would be in use[11].

2.2 *Lookup Keys and Indexes*

2.2.1 Flow Key Format

When the Dispatcher issues a command to the LUC it can use a format that includes a 116-bit *flow key*. The format of this flow key is protocol specific[12]. For TCP and UDP the 6-tuple of {IP Destination Address, IP Source Address, TCP Destination Port, TCP Source Port, IP Protocol, Receive Interface} is used, as shown in Figure 3. Note that the Receive Interface can either be the 802.1Q VLAN ID, the SPI-4 port or all zeros.

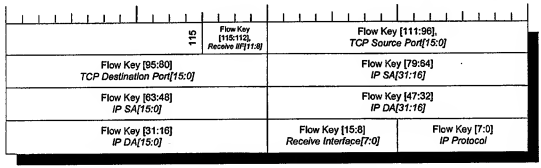

Figure 3: TCP Flow Key Format

2.2.2 Listen Key Format

As specified in section 1.2.6.1, if a match is not found on the flow key then some commands require the use of a listen lookup. This listen lookup uses various fields from the flow key, e.g. for TCP it might include the {IP Protocol, TCP Destination Port, IP Destination Address} fields. The LUC forms the listen key by applying

[11] See the *Flow Director CAM HLD* for details on how to ensure that only Workspace IDs 0, 4, 8 and 12 are used.

[12] In fact, the only time the LUC needs to know the exact bit locations of the flow key is when an access control list lookup is performed. If ACL is not supported for a protocol then the exact bit positions need not be known.

a 116-bit listen mask to the flow key. This listen mask is programmed into the LUC via the LISTEN_MASK registers of sections 4.8.19 through 4.8.22. Applying this mask to the flow key will yield a 116-bit listen key.

2.2.3 Socket ID Format

One of the features of the LUC is that it supplies a unique ID for each flow that it tracks. This ID is called the socket ID. If the LUC supports a maximum of N_{FLOWS} flows then the socket ID must range from 0 to (N_{FLOWS} – 1). For performance reasons the socket ID allows the LUC to directly access a flow's state: the socket ID maps directly into a flow memory address. Various LUC commands will then use this socket ID format as a method of describing a flow, e.g. during an update of a flow's state the Dispatcher supplies the socket ID: this should allow the LUC to avoid an expensive hash lookup.

The socket ID also allows the application on the host CPU to perform fast session lookups. For example, when the host CPU gets an event it can examine the socket ID and quickly determine the session information: this could be done using an array indexed by the socket ID, or by using a quick hash lookup on the socket ID.

2.2.4 Listen ID Format

The concept of a listen ID is similar to that of a Socket ID. The difference is that a listen ID is used to reference a socket that is in the listen state[13]. If the LUC supports a maximum of N_{LISTEN} listen entries then the listen ID must range from 0 to (N_{LISTEN} – 1).

2.3 *Hash Lookups*

Given the large number of flows that the LUC must maintain (up to 4 million), and the size of the flow key (116-bits), a reasonable implementation for the lookup mechanism would be hashing. Figure 4 illustrates, at a high level, the hash lookups that the LUC performs. The first step of a lookup is to take the 116-bit flow key, and hash it down to an N-bit wide value using a hash function. Using this N-bit hash value, we index into a hash table and find a hash entry. We then compare the key in the hash entry and the flow key: If the are the same then we have found a match, otherwise we must continue down the linked list of hash entries that have the same hash value. If a match with the flow key is found then a pointer in the hash entry points to the flow state data.

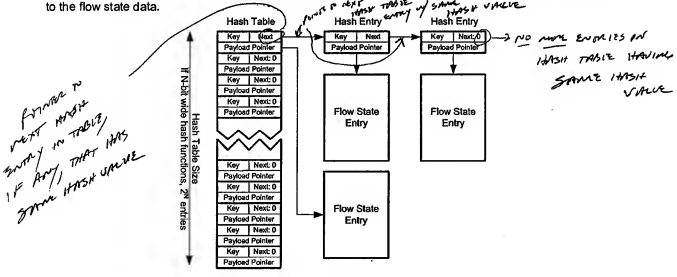

Figure 4: Hashing Structures

[13] In a regular socket API sockets are allowed to transition to and from the listen state by using the listen() function call. In the ACP this is not possible: once a listen socket (or regular socket) is created, it can be deleted but it cannot be modified.

2.3.1 Hash Table

The hash table is the table that is indexed by the hash value. It consists of an array of hash entries. Note that this is in contrast to some hashing implementations where the hash table simply contains pointers to a list of hash entries. The advantage of storing the first hash entry in the hash table is that it saves a level of indirection. However, this is at the expense of a much larger hash table and slightly complicated management[14]. For memory bandwidth reasons it was decided that the LUC hash table should contain hash entries rather than pointers.

Given that the hash table contains hash entries, each hash entry must have a valid bit. The reason is that a hash list may be empty, in which case the entry in the hash table is not valid.

2.3.1.1 Hash Table Size

The goal of a hashing algorithm is to reduce the search time for a key. For this reason we want the lists that come from the hash table to be as small as possible. This is achieved with good hashing functions and a wide hash value. A good rule to follow is that the hash table should be four times as large as the number of entries being stored. Since the number of entries in the LUC is flexible, the size of the hash table should also be flexible.

2.3.1.2 Hash Table Overflow

When a key is inserted into the hash table we must examine the valid bit of the first hash table entry. If this is valid then we have a collision, and we must link the new key into a list of hash table entries. The question is where do we get this new hash table entry?

One method of obtaining new hash table entries is to re-use an entry in the hash table. However, this method can be quite cumbersome to maintain. Another approach is to have a pool of available hash table entries that are used when we have a collision. This is the approach that the LUC uses, and the pool of available hash table entries is called the overflow hash table entries.

2.3.2 Hash Function

A core concept of a hashing algorithm is the hash function. This hash function must take an M-bit value and reduce it to an N-bit value. This N-bit value is then used to index the hash table.

There has been a considerable amount of research into the properties of various hash functions. Some of the hash functions that have been proposed in this research are relatively expensive to compute and yet only provide a small advantage over much simpler functions. For the LUC we use a simple XOR based hash function that folds the M-bit key into an N-bit value. This approach is reasonable so long as 2^N is large compared to the number of flows being supported. A good configuration to aim for is to ensure that 2^N is four times larger than the maximum number of keys (flows) to be supported.

As noted earlier, we must also allow for the hash table size to vary: for large numbers of flows (L7 switch) we need a large hash table, but for small numbers of flows (IP Storage) we can make do with a small table. The hash function must therefore also be configurable in terms of the value of N, the number of bits in the hash. The minimum number size of N is 17, allowing for 128K hash entries. The maximum size of N is 24, allowing for 16M hash entries. The LUC supports any value of N between 17 and 24.

The LUC hash function is computed as follows. First we chunk the flow key into four 24-bit values and one 20-bit value. The 20-bit value is {Flow Key[115:112], Flow Key[15:0]}, and the 24-bit values are {Flow Key[111:96], Flow Key[55:48]}, Flow Key[79:56], {Flow Key[95:80], Flow Key[23:16]} and Flow Key[47:24]. The selection of these bits from the flow key will become more apparent in section 2.3.2.1. Next Flow Key[79:56] is cyclically shifted by an amount S_1, where S_1 comes from a configuration register and has maximum value 16. Similarly, Flow Key[47:24] is cyclically shift by an amount S_2. Again, the reason for these shifts will become more apparent in section 2.3.2.1. These chunks are then folded into a single 24-bit

[14] When the very first hash entry is deleted we must copy the next hash entry into the table.

value using the XOR operator. We then reduce this 24-bit intermediate hash value into the real hash value by folding over the lower K-bits using the XOR operator, where a configuration register determines K. Figure 5 illustrates the hash function.

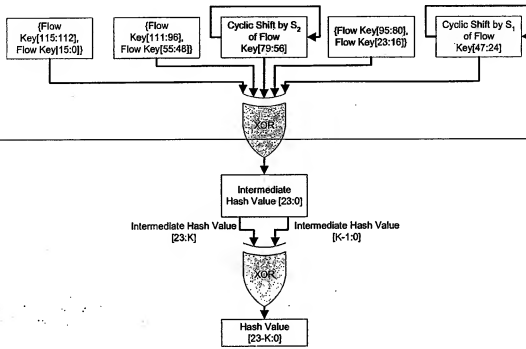

Figure 5: LUC Hash Function

2.3.2.1 TCP Hashing

From Figure 3 we can see that for TCP the flow key consists of the {Receive Interface, IP Protocol, TCP Source Port, TCP Destination Port, IP Source Address and IP Destination Address} tuple. The object of the hash function of Figure 5 is to generate a uniformly distributed hash value from this 6-tuple. However, there are a number of degenerate cases that we must be aware of:

1. Given that mega-proxies exist[15], the IP Source Address information could come from a small pool of IP addresses.
2. It could also be the case that all flows are going to the same service, i.e. the IP Destination Address and Destination Port could be the same.
3. We may only be terminating one protocol, so the Encoded Protocol field could be fixed.
4. In most configurations, the Receive Interface field will be fixed to a specific value, e.g. zero. Only transparent proxies will use different values in this field.

Note that traffic travels in both directions, so a source value in one direction is a destination value in another. It would seem that for TCP Termination we only see the traffic in one direction since we are the end point. However, a common application of TCP Termination is a proxy, so we still get to see the source and destination address information reversed.

In one direction the TCP Source Port and some number of bits of the IP Source address are the most variable information in the 116-bit flow key. In the other direction the TCP Destination Port and some number of bits of the IP Destination address are the most variable information. For this reason it is important

[15] An example is the Network Address Translation (NAT) that AOL does. Most of the AOL users appear on a small number of Internet addresses.

to make sure that either the {TCP Source Port, Selective IP Source Address Bits} or {TCP Destination Port, Selective IP Destination Address Bits} are spread across the N-bit hash value, and that they are not folded. In the following paragraphs we illustrate how the LUC hash function can ensure these values are well distributed in the hash value.

Figure 6 illustrates the layout of the 24-bit chunks that are XOR'ed together in Figure 5. Since the hash value is of configurable width, some number of least significant bits will be folded back into the new least significant bits. The number of bits folded can vary from none up to seven.

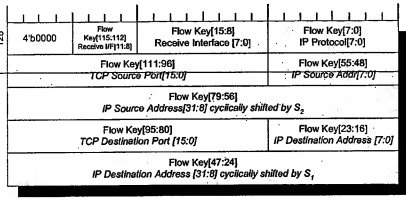

Figure 6: TCP Hash Chunks (24-bits)

By examining Figure 6 we can see that the TCP Source Port and TCP Destination Port fields are always spread across the hash value, even in the case when the lower seven bits are folded. We must also allow for some small number of bits varying in the IP source and destination addresses. In some cases these bits will be in the lower 8-bits of the IP source/destination address, in which case Figure 6 illustrates how this is butted against the TCP Source/Destination Port. However, it could be possible that the IP Source/Destination Address is varying the greatest in some other area rather than the lower 8-bits. This is the reason for the cyclic shifts of Figure 5: it allows us to position the most variable bits of the IP Source/Destination Address just to the right of the TCP Source/Destination Port.

2.3.2.2 Other Protocols

Section 2.3.2.1 demonstrated how the LUC hash function should perform well for a wide range of scenarios when TCP is the protocol being implemented. These same methods also apply to UDP. For other protocols a similar analysis should be done, and the various fields should be distributed in the flow key to ensure that the hash value is well distributed. Also note that if TACL is not being performed then the LUC need not know how the fields of the protocol are arranged in the flow key, therefore providing even more flexibility to ensure that the hash function performs well.

2.3.3 Hash Entry

With reference to Figure 4, the hash table consists of an array of hash elements. For clarity, the flow key has been shown in the TCP format. Each element must at least contain the key, a pointer to the payload, and a pointer to the next hash entry in the linked list. Figure 7 illustrates the exact format that the LUC uses for a B10 hash entry. The S10 hash format is exactly the same, except that it has an extra 32-bytes of spare at the end.

Table 3 defines the fields in this hash entry. For more information the on the design decisions for the hash entry, including its size, the reader is directed to section 5.1.

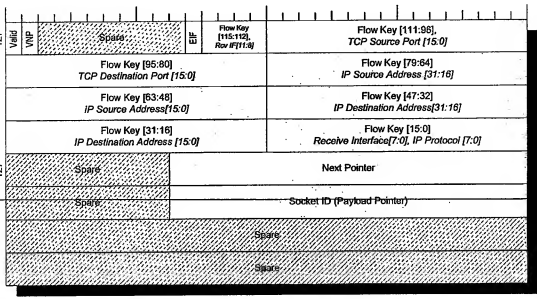

Figure 7: B10 Hash Entry Format

Field	Description
Valid	If set to 1 then this hash entry is valid, otherwise it is invalid. This is required since the initial hash table uses full hash entries rather than pointers. We must therefore be able to indicate that the hash entry in the hash table is not in use.
VNP	This is the Valid Next Pointer bit. If set to 1 then it indicates if the Next Pointer field has a valid value that we can traverse. A value of zero indicates that this entry is at the end of the hash list. It would have been possible to have Next Pointer = 0 indicate that this is the end of the list, but a single bit is easier to check.
EIF	This is the Entry is Flow bit. If set to 1 then this hash entry is for a flow key. If set to 0 then this hash entry is for a listen key. This allows us to distinguish between listen and flow keys when searching. The LUC sets this field depending on whether it is creating a flow or listen entry. Note that this field is not included in the hash.
Flow Key	The 116-bit bit key.
Next Pointer	This is an offset relative to the base of the Overflow Hash Table. It indicates where the next hash entry is in the list. Since this field is 22-bits, we are limited to 128MB of overflow hash table for the B10, and 256MB for the S10.
Socket ID	The Socket ID is actually the payload pointer. It is an index of a flow state in the Flow State area of memory. By shifting the Socket ID by the appropriate amount, and adding the base address for the flow state table, it is possible to turn this into a DDR address.

Table 3: Hash Entry Field Descriptions

2.3.4 Hash Maintenance

2.3.4.1 Hash Searching

The process of searching for a hash key is simple:

1. The hash of the key is taken according to section 2.3.2
2. Using the hash value, the hash entry in the hash table is fetched. If the hash entry is not marked as valid, then the search is terminated: the flow key is not present.

3. If the hash entry is marked as valid then we examine the EIF flag. If we are searching for a flow key, and the EIF flag is 0, then there is no match and we continue with step 5 below. Similarly, if we are searching for a listen key and the EIF flag is 1 then there is no match and we continue with step 5 below.
4. The flow key in this hash entry is compared against the flow/listen key being searched. If a match is found then we stop, otherwise we continue below.
5. The Valid Next Pointer flag of the hash entry is examined. If it is not set then the search is terminated; the flow/listen key is not present.
6. The next hash entry, as indicated by the Next Pointer is fetched. The flow key of that entry is compared against the search flow/listen key again allow for the EIF flag. If a match is found then we stop, otherwise we continue at step 5.

2.3.4.2 Hash Entry Insertion

Hash entry insertion is the process of inserting a hash entry into an existing chain. Such entries can be added at the end of the hash list, at the front, or anywhere in the middle. The only important thing is that the LUC correctly set the EIF flag depending on whether it is inserting a flow key hash or a listen key hash.

2.3.4.3 Preventing Long Hash Chains

It can be seen that very long hash chains cause a significant degradation in performance. In fact, not only is the search for that particular flow key slow, but it also slows down the searches for other flow keys since the DDR memory is a shared resource. Such a long hash chain could be a Denial of Service attack.

The LUC prevents hash chains from getting too long by refusing to create new entries when the list reaches a predetermined length. The MAX_HLEN field of the HASH_PARAMS register (section 4.8.23) specifies the maximum length of a hash chain. If creating a new hash entry would cause the hash list to become longer than this length then an entry is not created. For more information on the implementation of this feature the reader is directed to sections 3.5.13 and 3.5.14.

2.3.4.4 Hash Entry Removal

Removal of a hash entry may require that the hash table be overwritten. This occurs when the entry being removed is the entry in the hash table. In that case the next element, if it exists, must be copied into the hash table.

Removing entries from anywhere other than the hash list head simply requires careful maintenance of the *Next Pointer* in the hash entry.

2.3.5 Flow State Entry

2.3.5.1 Flow State Entry Format

With reference to Figure 4, a flow state entry is the data pointed to by the payload pointer of the hash entry. In the case of the LUC this is the data pointed to by the Socket ID. The size of this data is set at configuration time to be the range 128 to 2048 bytes. Figure 8 illustrates the format of this data. The first nine 32-bit words are reserved by use by the LUC. When a flow state is sent to a Processor Core the LUC will overwrite these words with the workspace header, as illustrated by the vertically shaded fields.

When the LUC deposits this workspace back in the DDR memory it may write any values in the vertically shaded areas since they are not used. Note that the flow key is specifically indicated in this figure. The reason for this is that the LUC will store the flow key in this position so that when a timer expires it can find out what the flow key is for that timer. The remaining space in the flow state is used to store the software formatted flow state.

The reason for keeping the flow state entry and the hash entry separate is because the hash table contains complete entries, not pointers. If we were to store the flow state and hash entry together then the hash table would be enormous.

Only the LUC interprets the flow state memory using Figure 8. When this is passed to the protocol core as a workspace it has extra information embedded into it (see Figure 42 later in this document).

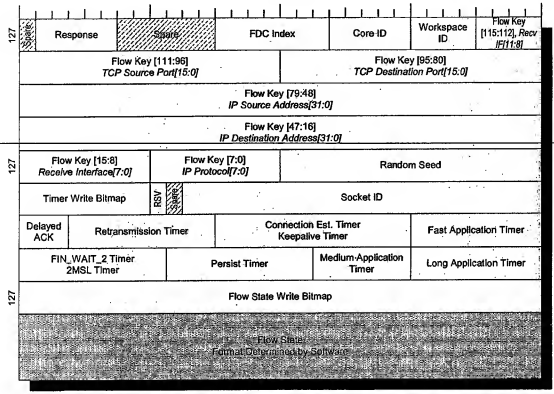

Figure 8: Flow State Entry Format

2.3.5.2 Selective Flow State Write Backs

The *Flow State Write Bitmap* of Figure 8 is used to selectively write back portions of the flow state. The entire area of Figure 8 is split into 64-byte chunks, with the first chunk starting at the first word of the flow state. Since the maximum workspace size is 2KB, there will be at most 32 64-byte chunks. If bit i of the *Flow State Write Bitmap* is set then chunk i should be written back to DDR memory. Note that the first chunk, with i = 0, must be written back when a flow is created. See sections 2.3.5.3.1 and 2.3.5.4.1 for details on who (Protocol / Interface Core) is allowed to write this area.

Selective write backs not only allow us to decrease the memory bandwidth requirements, but they also allow flow state splitting. This is described in more detail in the sections below.

2.3.5.3 Exclusive Flow State Splitting

As discussed in section 1.2.5 on split Protocol Core and Interface Core processing, it is possible to split a flow state across two workspaces. In this section we describe how this splitting is performed. We assume that no sharing of the workspace is being performed. Details on how to share workspace areas are presented in section 2.3.5.4.

Figure 9 illustrates the splitting of a flow state. It shows the three areas where the state is held:

1. When it is held in the DDR memory of the LUC, under the format of Figure 8.

2. When it is in a Protocol Core's workspace.
3. When it is in an Interface Core's workspace.

Note that the exact same layout exists for listen entries, but in this case LISTEN_SIZE and LIS_PC_SIZE are the parameters used.

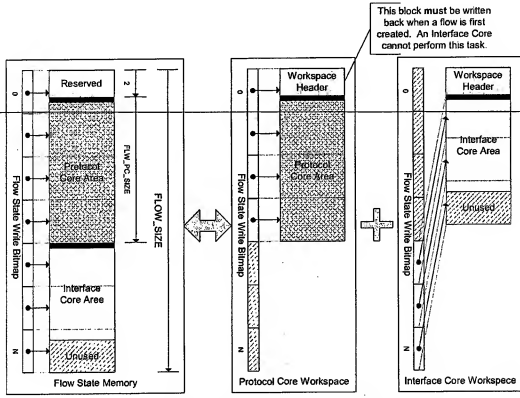

Figure 9: Exclusive Flow State Splitting

2.3.5.3.1 *Flow State Memory Parameters*

With reference to Figure 9 and the flow state memory, the flow state is split into the following regions in this order:

1. A reserved area where the LUC will write the first two 128-bit words of the workspace header. It corresponds to the first two 128-bit words of Figure 8.
2. An area for the *Flow State Write Bitmap* of Figure 8. This is illustrated in the diagram as a black area. This is 32-bits in size.
3. The *Protocol Core Area*, an area for the exclusive use of a Protocol Core.
4. Another black area that will be used for the *Flow State Write Bitmap* of the Interface Core workspace header.
5. The *Interface Core Area*, an area for the exclusive use of an Interface Core.

The FLOW_SIZE field of the LUC_PARAMS register (see section 4.8.9) defines the size of the flow state in the DDR memory. This is specified as an enumerated set of 128, 256, 512, 1024 or 2048 bytes.

As described in section 2.3.5.2 on selective write backs, the *Flow State Write Bitmap* is relative to the very first byte of the flow state held in DDR memory.

When exclusive flow state splitting is used, the Protocol Core **must** write back the very first block when a flow is created. The reason for this is that the first block contains the Flow Key, as illustrated in Figure 8. The LUC does not write this Flow Key, but instead relies on the Protocol Core to write it. The Interface Core cannot perform this task since the Interface Core workspace header is not written back to the flow state. See section 2.3.5.6 for the full set of rules regarding workspace write backs.

2.3.5.3.2 Protocol Core Workspace Parameters

When the LUC sends a split flow state to a Protocol Core, it configures the workspaces as shown in the *Protocol Core Workspace* block of Figure 9. This is a direct copy of the first (2 + FLW_PC_SIZE) 128-bit words from the DDR memory into the workspace. Note however that the LUC overwrites the first two 128-bit words with the Workspace Header.

~~The FLW_PC_SIZE field of the LUC_PARAMS register defines the size of the *Protocol Core Area*.~~ Note that FLW_PC_SIZE includes the 32-bit black area that is reserved for the *Flow State Write Bitmap* of Figure 8. This means that the number of bytes that the Protocol Core can use is (((FLW_PC_SIZE * 16) – 4). Also note that (FLW_PC_SIZE + 2) when expressed in bytes must be an exact multiple of 64-bytes. The same applies to listen entries (LIS_PC_SIZE).

The *Flow State Write Bitmap* for a Protocol Core requires no special handling. The reason is that the Protocol Core area comes directly from the DDR memory, and as such the *Flow State Write Bitmap* exactly lines up. Later in this document (section 2.3.5.7) we describe how an Interface Core can be prevented from modifying the *Protocol Core Area*.

2.3.5.3.3 Interface Core Workspace Parameters

When the LUC sends a split flow state to an Interface Core, it configures the workspaces as shown in the *Interface Core Workspace* block of Figure 9. To create this workspace the LUC does two things:

1. It places the same workspace header that it did for the Protocol Core at the front of the workspace. This will occupy the first two 128-bit words. Note that when we say *Workspace Header* we do not include the 32-bit *Flow State Write Bitmap*, since the LUC does not set this field.
2. The LUC then copies the *Interface Core Area* from the DDR memory and places it just after the workspace header, i.e. it starts writing the *Interface Core Area* in the second 128-bit word of the Interface Core workspace. The first 32-bits of this area are unusable by the Interface Core since it contains the *Flow State Write Bitmap*.

The size of the *Interface Core Area* is computed by using FLW_PC_SIZE and FLOW_SIZE. As illustrated, the size of the *Interface Core Area* is (FLOW_SIZE – FLW_PC_SIZE – 2) 128-bit words. Again, this includes the 32-bit black area that is reserved for the *Flow State Write Bitmap* of Figure 8. This means that the number of bytes that the Interface Core can use is (((FLOW_SIZE – FLW_PC_SIZE – 2) * 16) – 4).

The *Flow State Write Bitmap* for an Interface Core requires some level of special handling. The reason is that bit i of the *Flow State Write Bitmap* does not correspond to the i'th 64-byte chunk of the *workspace*, but instead refers to the i'th 64-byte chunk of the flow state in DDR memory. The Interface Core must therefore be aware of how large the Protocol Cores flow state is, and then set the appropriate bits in its *Flow State Write Bitmap*. Later in this document (section 2.3.5.7) we describe how a Protocol Core can be prevented from modifying the *Interface Core Area*.

2.3.5.3.4 Restrictions

Note how in Figure 9 the *Protocol Core Area* ends on a 64-byte boundary. This is a strict requirement of the LUC since ending on anything other than a 64-byte boundary would cause problems where the Protocol Core can modify the Interface Cores area or vice-versa. If the *Protocol Core Area* does not end on a 64-byte boundary then it **must** be padded so that it does.

2.3.5.4 Flow State Splitting with Sharing

The above section defined how a flow state is exclusively split across a Protocol Core and an Interface Core. In this section we slightly modify that description to allow for an area of flow state that is shared by both cores. Using a shared region needs careful thought, as will be described in later sections.

Figure 10 below illustrates the same elements of Figure 9 on exclusive flow state splitting, except this time a shared area has been included.

When flow state splitting with sharing is used, either the Protocol Core or the Interface Core **must** write back the first block when a flow is created. In the case of shared flow state splitting then this first block will be in the *Shared Area*. In that case either the Protocol Core or the Interface Core **must** write back the first chunk of the *Shared Area*, even if that area has not changed. This is for the exact same reasons explained at the end of section 2.3.5.3. See section 2.3.5.6 for the full set of rules regarding workspace write backs.

Note that the exact same layout exists for listen entries, but in this case LISTEN_SIZE, LIS_PC_SIZE and LIS_SH_SIZE are the parameters used.

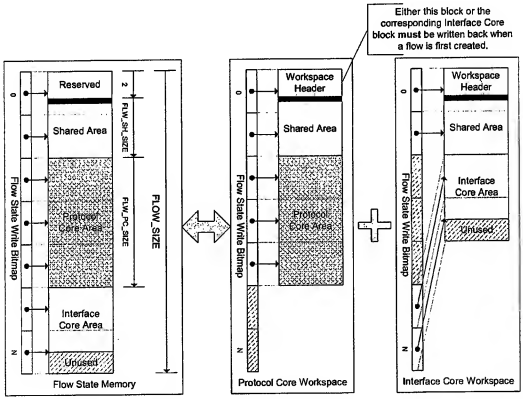

Figure 10: Flow State Splitting with Sharing

2.3.5.4.1 Flow State Memory Parameters

The first thing we note is that the *Shared Area* is the first portion of the flow state in the DDR memory. Note that since the *Shared Area* is the first area, it has to take the penalty of the 32-bits reserved for the *Flow State Write Bitmap*. The rest of the flow state in DDR memory is exactly the same as the exclusive splitting case.

2.3.5.4.2 Protocol Core Workspace Parameters

When the LUC sends a split flow state that contains sharing to a Protocol Core, it configures the workspaces as shown in the *Protocol Core Workspace* block of Figure 10. This is a direct copy of the first (2 + FLW_SH_SIZE + FLW_PC_SIZE) 128-bit words from the DDR memory into the workspace. This direct copy includes the entire *Shared Area*. Note however that the LUC overwrites the first two 128-bit words with the Workspace Header. Note that (FLW_SH_SIZE + 2) when expressed in bytes must be an exact multiple of 64-bytes. The same applies to listen entries (LIS_SH_SIZE).

As with the exclusive case, the FLW_PC_SIZE field of the LUC_PARAMS register defines the size of the *Protocol Core Area*. However, unlike the exclusive splitting case, the FLW_PC_SIZE does not include 32-bits of *Flow State Write Bitmap* for the shared case, since this is included in the *Shared Area*. Note that FLW_PC_SIZE when expressed in bytes must be an exact multiple of 64-bytes. The same applies to listen entries (LIS_PC_SIZE).

The *Flow State Write Bitmap* for a Protocol Core requires no special handling. The reason is that the Protocol Core area comes directly from the DDR memory, and as such the *Flow State Write Bitmap* exactly lines up. Later in this document (section 2.3.5.7) we describe how an Interface Core can be prevented from modifying the *Protocol Core Area* or *Shared Area*.

2.3.5.4.3 Interface Core Workspace Parameters

When the LUC sends a split flow state that contains sharing to an Interface Core, it configures the workspaces as shown in the *Interface Core Workspace* block of Figure 10. To create this workspace the LUC does three things:

1. It places the same workspace header that it did for the Protocol Core at the front of the workspace. This will occupy the first two 128-bit words. Note that when we say *Workspace Header* we do not include the 32-bit *Flow State Write Bitmap*, since the LUC does not set this field.
2. The LUC copies the *Shared Area* from the DDR memory and places it just after the workspace header, i.e. it starts writing the *Interface Core Area* in the second 128-bit word of the Interface Core workspace. The first 32-bits of this area are unusable by the Interface Core since it contains the *Flow State Write Bitmap*.
3. The LUC then copies the *Interface Core Area* from the DDR memory and places it just after the *Shared Area*.

The size of the *Interface Core Area* is computed by using FLW_PC_SIZE, FLW_SH_SIZE and FLOW_SIZE. As illustrated, the size of the *Interface Core Area* is (FLOW_SIZE – FLW_PC_SIZE - FLW_SH_SIZE– 2) 128-bit words. Unlike the case with exclusively split flow states, the *Interface Core Area* for a split that includes a *Shared Area* does not require 32-bits for the *Flow State Write Bitmap*. Instead the full *Interface Core Area* is available.

The *Flow State Write Bitmap* for an Interface Core requires even more special handling when a shared region is present. For the *Shared Area* of the workspace in the Interface Core, it is mapped directly to the corresponding area in the DDR memory. Therefore, the first (2 + SH_SIZE) / 4 bits of the *Flow State Write Bitmap* will correspond to the *Shared Area* of the workspace. The *Interface Core Area* is then treated in exactly the same fashion that it was when a shared region was not present, i.e. some level of shifting is required. Later in this document (section 2.3.5.7) we describe how a Protocol Core can be prevented from modifying the *Interface Core Area* or *Shared Area*.

2.3.5.5 Shared Area Issues

There are a number of issues that the software must be architected for when shared areas are in use. These include:

1. If the *Shared Area* is modified, then the other Processor Core will not see that modification until all outstanding events have been processed. The reason is that the workspace is not written back to

the DDR memory until all the outstanding events have been processed. Only when it is written back to DDR memory will it have the opportunity to be re-split.
2. The software must control who writes the shared area. The LUC provides no mechanism for supporting this. If two Processor Cores write the same shared area then the flow state in DDR memory will contain the data from the Interface Core.

These issues make it clear that the software must be architected very carefully in order to take advantage of workspace sharing. However, if done properly it does allow reduced Processor Core cycle counts, and reduced DDR memory bandwidth[16]. One way to successfully use this area is to configure a *Shared Area* a listen entry that matches the *Shared Area* of a flow entry. Therefore, when a flow entry is created both the Interface Core and Protocol Core get to see the same shared values that are never changed.

2.3.5.5.1 Restrictions

Note how in Figure 10 the *Shared Area* and *Protocol Core Area* end on a 64-byte boundary. This is a strict requirement of the LUC. The reason for this is exactly the same as those described in section 2.3.5.3.4 above. If the *Shared Area* does not end on a 64-byte boundary then it must be padded so that it does. For the same reasons, the *Protocol Core Area* must also be padded to end on a 64-byte boundary.

2.3.5.6 Workspace Write Back Rules

The following rules must be obeyed when cores issue a done event in response to an input event with a workspace:

1. The workspace must be written back before the done event is sent to the Dispatcher.
2. For done events that are to update the workspace, either the Protocol Core or the Interface core must write back at least the workspace header[17]. If this is not the first update for this flow then the writeback bitmap can be set to all zeros. Note that it is acceptable for both the Protocol Core and the Interface Core to write back a workspace
3. For done events that are to tear down a flow, it is not required that the Protocol Core or Interface Core write back a workspace[18] (or workspace header). If a workspace is written back then it is ignored.
4. For flows that are created, the protocol core must write back at least the first 64-bytes of a workspace, and the corresponding bit must be set in the writeback bitmap. In is invalid for the Interface Core to only write back the workspace, i.e. the Protocol Core must do it (Interface Core may also write back but that is optional).

2.3.5.7 Memory Protection

Given that two independent Processor Cores can potentially modify an area of the flow state, it was regarded as important that we perform some level of memory protection. This is even more important when you consider that the Interface Core may be running customer code whose source is not available for examination.

The LUC offers this memory protection by using simple masks. Before a *Flow State Write Bitmap* is examined, we first apply one of four masks:

- One mask is used by a Protocol Core to apply to a *Flow State Write Bitmap* for a flow state (see the register in section 4.8.59).

[16] If *Shared Area* of a workspace was not available then the information would simply be replicated, possibly causing an increased flow state size. Similarly, replicating this information will consume processor cycles.
[17] This is required since the LUC must fetch the Socket ID from the workspace header. Without this Socket ID it cannot manage the Timer Table correctly.
[18] On a teardown the LUC obtains the Socket ID from the hash table entry, so it does not need the workspace header.

- Another mask is used by a Protocol Core to apply to a *Flow State Write Bitmap* for a listen state (see the register in section 4.8.60).
- Another mask is used by an Interface Core to apply to a *Flow State Write Bitmap* for a flow state (see the register in section 4.8.61).
- The final mask is used by an Interface Core to apply to a *Flow State Write Bitmap* for a listen state (see the register in section 4.8.61).

By correctly setting these global masks, a Protocol or Interface Core can prevent its region, or a *Shared Area* from being overwritten.

2.3.6 Listen State Entry

Listen state entries serve the same purpose as flow state entries, except that they are smaller. They hold state information for connections that are in the listen state, i.e. are listening on a virtual IP address for connections to accept. This data is the same format as Figure 8.

2.3.7 Flow Key and Listen Key Hash Table Sharing

During a lookup the LUC can use two types of keys: a flow key and a listen key. We could keep two separate hash tables for these keys, but that would require a much larger house keeping overhead. Instead the flow key and listen key share the same hash table. As described in section 2.2.2, applying the listen mask to the flow key forms the listen key[19]. For most customers, the TCP listen mask will mask out the Receive Interface, IP source address and the TCP source port. There is still a problem in that it could be possible that a listen key and flow key have the same value – we must be able to distinguish them.

As shown in Figure 7 of the Hash Entry Format, each entry has an EIF flag. This is the Entry Is Flow flag, and is set to 1 for flow keys, and 0 for listen keys. When the LUC is searching for a flow key it only inspects hash entries whose EIF flag is set to 1. When the LUC is searching for a listen key it only inspects hash entries whose EIF flag is set to 0. This avoids the problem where a flow key and listen key may have the same value.

2.3.8 Hashing Statistics

In order to allow tuning of the hash function, we must keep some number of statistics on the hash lookup performance. However, these statistics must not be overly burdening in turns of computation or memory. The hashing statistics for the LUC are:

1. A running total of the number of flow key and listen key mismatches. See sections 4.8.44 and 4.8.45 for more details.
2. Each LUC Engine records the maximum hash list length is has encountered during a search. See sections 4.8.46 through 4.8.53.

2.4 LUC Commands Overview

2.4.1 LUC Commands

Table 4 defines the LUC commands and any associated flow locking that is required. Flow locking is described later in this document in section 3.2. For now it is sufficient to say that the FDC guarantees that commands for the same **flow key** are processed one at a time. However, these commands may have the same **hash value**, so the flow locker is used to ensure that two or more flows with the same hash value do not interfere with each other.

[19] For some commands, e.g. LLKC, the listen key is taken directly from the flow key without applying the listen mask. The reasons for this will become more apparent in later sections.

Command	Value	Description	Flow Locking Required
LFK	5'h0	Lookup with Flow Key, do not create if not found.	Read Lock on Flow Hash
LFLKC	5'h1	Lookup with Flow Key. If not found then lookup with Listen Key: create an entry if Listen Key is found.	If the flow hash != listen hash Ganged: Write Lock on Flow Hash,Read Lock No Yield on Listen Hash Else Write Lock on Flow Hash
USID	5'h2	Update with Socket ID.	None
ULID	5'h3	Update with Listen ID.	None
TDLK	5'h4	Teardown with Listen Key.	Write Lock on Listen Hash
TDFK	5'h5	Teardown with Flow Key.	Write Lock on Flow Hash
LFKC	5'h6	Lookup with Flow Key, create entry if not found.	Write Lock on Flow Hash
LLKC	5'h7	Lookup with Listen Key, create entry if not found.	Write Lock on Listen Key
LFLK	5'h8	Lookup with Flow Key. If not found then lookup with Listen Key: do not create an entry.	If the flow hash != listen hash Ganged: Write Lock on Flow Hash, Read Lock No Yield on Listen Hash[20] Else Write Lock on Flow Hash
LSID	5'h9	Lookup with Socket ID.	None
RELS	5'ha	Release a workspace	None

Table 4: LUC Commands

As will be described later in this section, for each lookup command in Table 4 a Processor Core must trigger either an update, a tear down or a release. This could be an Update with Socket ID (USID), Update with Listen ID (ULID), Teardown with Flow Key (TDFK), Teardown with Listen Key (TDLK) or Release (RELS) LUC command. The exact LUC command that must be used depends on the response that the LUC gave in the *Response* field of the Workspace Header. There are only one or two LUC commands that can be used for each *Response* value. Table 33 on *Workspace Response Values* defines the complete set.

2.4.2 Lookup with Flow Key (LFK)

The Lookup with Flow Key command takes a 116-bit flow key as a parameter and attempts to find that flow state in the LUC. Note that due to the *Entry Is Flow* bit of a hash entry (see Figure 7), it is not possible for this flow key to be mistaken for a listen key. If the flow key is found then the flow's state is sent to the appropriate Processor Cores with a *Flow Found* response. If the flow key is not found then a workspace (header only) with a *Not Found* response is sent to the Processor Cores.

2.4.3 Lookup with Flow Key and Create (LFKC)

The Lookup with Flow Key and Create command performs the same task as the LFK command. The difference occurs when the flow key is not found. If the flow key is not found for an LFKC command, then a new flow state entry is created with that exact flow key. A workspace (header only) is then sent to the Processor Cores with a *Flow Created* response.

Note that it is possible that the flow state entry could not be created due to lack of LUC resources. In this case a workspace (header only) with a *No Space* response is sent to the Processor Cores. It is also possible that the hash list length would exceed its maximum value, or that the flow was not created due to a Termination Access Control List (TACL) denial (future sections describe TACL). These are indicated by *Hash Too Long* and *TACL Denied* responses.

[20] A ganged read lock on flow hash and read lock on listen hash would be adequate for the LFLK command. However, to simplify the flow locker design we use a ganged write lock on flow hash, read lock on listen hash.

2.4.4 Lookup with Flow Key, then Listen Key (LFLK)

The Lookup with Flow Key, then Listen Key command first looks up the flow key in the LUC hash table. If it is found then that flow state is sent to the appropriate Processor Cores with a *Flow Found* response. If the flow key is not found then a listen key lookup is performed. The listen key is formed from the flow key by applying a mask. Note that the listen key can then be further modified by the result of a Termination Access Control List (TACL) match. If a listen key is found then the listen state is reported to the appropriate Processor Cores with a *Listen Found* response. If the listen key is not found then a workspace (header only) with a *Not Found* response is reported to the Processor Cores.

2.4.5 Lookup with Flow Key, then Listen Key and Create (LFLKC)

The Lookup with Flow Key, then Listen Key and Create command performs the same tasks as the LFLK command above. The difference comes when the listen key is found. If a listen key is found then the LFLKC commands creates a new flow state entry with the flow key, and then reports the listen state to the appropriate Processor Cores with a *Listen Found, Flow Created* response.

Since this command is creating flow state entries, it can also respond with the same *No Space*, *Hash Too Long* and *TACL Denied* responses described for the LFKC command.

2.4.6 Lookup with Socket ID (LSID)

The Lookup with Socket ID command takes a 22-bit Socket ID and fetches the flow state associated with it. Note that this does not require a search of the hash table: it is a direct memory access. For this reason it is therefore more efficient than the lookup commands that use a flow key, and should be used where ever the Socket ID is available.

The only response from this command is *Flow Found*. It is not possible for the LUC to detect whether a valid flow state exists for this Socket ID[21].

2.4.7 Lookup with Listen Key and Create (LLKC)

The Lookup with Listen Key and Create command takes a 116-bit bit listen key and looks it up in the hash tables. It should be noted that the listen mask is not applied to the flow key to form a listen key. Instead the listen key is taken directly from the flow key of the LLKC command. This is required so that listen entries can be created that have bits set outside of the listen mask, which is needed for the operation of the *Listen Fixer* described later in the section on Termination Access Control Lists (TACL). Note that due to the *Entry Is Flow* bit of a hash entry (see Figure 7), it is not possible for this listen key to be mistaken for a flow key.

If the listen key is found then the listen state is sent to the appropriate Processor Cores using a *Listen Found* response. If a listen entry is not found then one is created, using the listen key that was searched for, i.e. not using a masked listen key. A workspace (header only) is then sent to the Processor Cores using a *Listen Created* response.

Since this command is creating flow state entries, it can also respond with the same *No Space*, *Hash Too Long* and *TACL Denied* responses described for the LFKC command.

2.4.8 Update with Socket ID (USID)

The Update with Socket ID command is used to update the flow state of a given Socket ID. The Processor Cores direct the Dispatcher to send this command by using the appropriate Event Type in a Done Event. Using the *Flow State Write Bitmap* it is possible for a Processor Core to indicate which chunks of the flow state need to be updated. Reducing the amount of state that has to be updated increases the performance

[21] It would be possible to detect an invalid Socket ID if the LUC wrote an *Invalid Flow State* marker to a flow's state when it was deleted. However, this would require an extra memory transaction when a flow is torn down, so in the interests of efficiency it is not performed.

of the LUC. The Processor Cores also use the *Timer Write Bitmap* to indicate which timers should be overwritten.

After the flow state has been updated, the LUC uses the Message Bus to mark the appropriate workspace IDs as invalid. It then attempts to remove the flow entry from the Flow Director CAM (FDC) using the RMFIDX command. If the removal is successful then the LUC considers the flow checked in, i.e. no longer being processed. If the FDC entry is not removed, then it must mean that another event has arrived for that flow. In that case the LUC re-marks the appropriate workspace IDs as valid and continues to classify the flow as checked out, i.e. being processed by a pair of Processor Cores. Note that it is important that the LUC does not re-send the workspace to the Processor Cores, since their version is more up to date.

There is no response from this command.

2.4.9 Update with Listen ID (ULID)

The Update with Listen ID command is exactly the same as the USID command, except that a listen state is updated using a Listen ID. There are also no timers associated with a listen state.

2.4.10 Teardown with Flow Key (TDFK)

The Teardown with Flow Key command is used to remove an entry from the hash table. The Processor Cores direct the Dispatcher to send this command by using the appropriate Event Type in a Done Event. The LUC searches for the supplied 116-bit key in the hash table. If it is found then it is removed and all resources associated with it are de-allocated. If it is not found then an error bit is set in the status register.

After the flow has been torn down, the LUC uses the Message Bus to mark the appropriate workspace IDs as invalid. It then attempts to remove the flow entry from the Flow Director CAM (FDC) using the RMFIDX command. We know, due to the operation of the FDC, that if a TDFK command has been issued then the FDC entry is in the *DELETE* state. FDC entries that are in the *DELETE* state cause the Dispatcher to stall processing of that entry, effectively waiting for the LUC to finish tearing down the flow. Therefore, if the FDC entry is not removed, then an error condition has occurred and the LUC sets a bit in the status register.

There is no response from this command.

2.4.11 Teardown with Listen Key (TDLK)

The Teardown with Listen Key command follows the same steps as the TDFK command described above. The difference is that it is used to remove a listen entry from the hash table rather than a flow entry. Note that as with the LLKC command, and for the same reasons, the TDLK command uses the full 116-bit flow key as the listen key. The LUC searches for the supplied 116-bit key in the hash table. If it is found then it is removed and all resources associated with it are de-allocated. If it is not found then an error bit is set in the status register.

The TDLK command follows the same interaction with the Flow Director CAM (FDC) as the TDFK command.

There is no response from this command.

2.4.12 Release (RELS)

The Release command is used for cases when a flow or listen key has been looked up, but either:

1. The key was not found, i.e. a *Not Found* response.
2. There was no space to allocate an entry, i.e. a *No Space* response.
3. An entry was not created since the hash list would have become too long, i.e. a *Hash Too Long* response.
4. An entry was not created since it was denied by a Termination Access Control List (TACL) entry, i.e. a *TACL Denied* response.

In any of the above cases we need the LUC to mark the appropriate workspace IDs as invalid, and de-allocate any resources in the Flow Director CAM (FDC). This is done using the Release command.

First the LUC uses the Message Bus to mark the appropriate workspace IDs as invalid. It then attempts to remove the flow entry from the Flow Director CAM (FDC) using the RMFIDX command. We know, due to the operation of the FDC, that if a RELS command has been issued then the FDC entry is in the *DELETE* state. FDC entries that are in the *DELETE* state cause the Dispatcher to stall processing of that entry, effectively waiting for the LUC to finish releasing the flow. Therefore, if the FDC entry is not removed, then an error condition has occurred and the LUC sets a bit in the status register.

There is no response from this command.

2.5 Termination Access Control Lookups

As described in section 1.2.6.2, the LUC provides an access control mechanism similar to Access Control Lists of routers. This security mechanism allows an access control list to be traversed before a connection is terminated: Termination Access Control Lists (TACLs).

2.5.1 When To Apply TACL

To improve performance, the LUC only searches the TACL table when a flow key is not found. The reasoning behind this is that if a flow key is found then we must have already performed a TACL search and found an *allow*, otherwise it would not have been created[22]. We therefore only perform TACL lookups for LUC commands that can create flow entries. These are described below. Note that for each of the three commands described below, TACLs can be enabled or disabled individually (see the CONFIG register of section 4.8.6). Also note that if a TACL lookup does not find a match then the *Default Allow/Deny* field (DEF_AD) of the Configuration register (section 4.8.6) is used. This effectively means that a TACL lookup will *always* respond with an allow or deny.

2.5.1.1 LFKC TACL

For a Lookup Flow Key Create (LFKC) command we first search for a flow key. If a flow key is not found, and TACL lookups are enabled for the LFKC command, then a TACL lookup is performed. If the result of this TACL lookup is a deny, then a TACL denied response is reported to the Processor Cores, and no flow key entry is created. If the result of the TACL lookup is an allow, then a flow key entry is created as normal.

2.5.1.2 LFLKC TACL

For a Lookup Flow Key Listen Key Create (LFLKC) command we first search for a flow key as normal. If a flow key is not found, and TACL is enabled for the LFLKC command, then we next perform a TACL lookup. If the result of this TACL lookup is a deny, then a TACL denied response is reported to the Processor Cores, and no listen key search is performed. If the result of the TACL lookup is an allow, then a listen key search is performed as normal.

2.5.1.3 LLKC TACL

The Lookup Listen Key Create (LLKC) command is processed in exactly the same fashion as the LFKC command above, i.e. if a listen key is not found then we use the TACL to determine whether one can be created. For most scenarios, it is likely that TACL lookups for LLKC commands will be disabled.

[22] During runtime, if the TACL is changed then it would be possible that existing flows would not find a TACL allow, and would therefore not have been terminated. The software must detect this condition and take appropriate action, e.g. delete the flow entry.

2.5.2 TACL Entry Format

Figure 11 illustrates the format of a B10 TACL entry in DDR memory. For the S10 LUC the format of a TACL entry is exactly the same, except that two TACL rules are contained in a single 64-byte DDR burst. The fields of Figure 11 are further explained in Table 5.

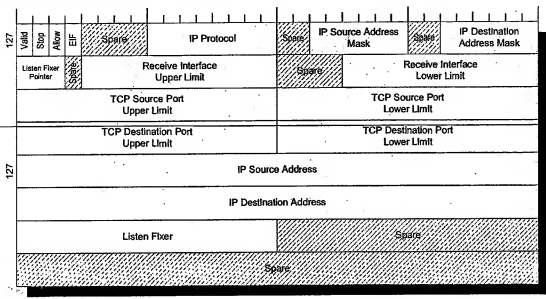

Figure 11: B10 TACL Entry Format

Field	Description
Valid	If set to 1 then this is a valid TACL rule. If set to 0 then it is an invalid rule that should not be compared against. This is used to mark entries as invalid instead of having to completely remove them from the TACL table. Note that even if the Valid bit is 0, the Stop field defined below is still interpreted, i.e. the last element in a table could be an invalid TACL entry with the stop bit set.
Stop	This flag is used to mark the end of the TACL table. If this flag has value 0, and if this TACL rule does not match, then the next TACL rule will be fetched. If this flag has value 1, and if this TACL rule does not match, then the search will stop and the default allow/deny bit (DEF_AD) of the CONFIG register will be used (see section 4.8.6).
Allow	This is the Allow / Deny bit. This is only used if a match is found against this rule. Value 1 indicates that the flow should be allowed, otherwise the flow should be denied.
EIF	This is the *Entry Is Flow* flag. If this entry is set to value 1 then the ACL rule will only be matched against for the LFKC and LFLKC commands, i.e. flow key searches. If this entry is set to value 0 then the ACL rule will only be matched against for LLKC commands, i.e. listen key searches.
IP Protocol	This is the exact match IP protocol value to match against the flow key.
IP Source Address Mask	This is an encoding of the IP Source Address Mask to use. This value is the number of 1's that are in the mask, counting from the MSB to the LSB. For example, if this field has value 0 then the mask is 0.0.0.0. If it has value 8 then the mask is 255.0.0.0. If it has value 32 then the mask is 255.255.255.255. Values above 32 are not valid.
IP Destination Address Mask	This is an encoding of the IP Destination Address Mask to use. It uses the same encoding as the IP Source Address Mask defined above.
Receive Interface Upper Limit	This is the upper limit of a receive interface that we will match against. It is inclusive, i.e. up to and including this value.
Receive Interface Lower Limit	This is the lower limit of a receive interface that we will match against. It is inclusive, i.e. greater than or equal to this value.

Field	Description
Listen Fixer Pointer	This 3-bit field indicates where in the listen key the Listen Fixer value should be written if this TACL rule matches. Note that the Listen Fixer value can only be written to a 16-bit aligned area of the listen key. If the Listen Fixer Pointer has value 0 then Listen Key[15:0] is overwritten with the Listen Fixer. If the Listen Fixer Pointer has value 1 then Listen Key[31:16] is overwritten and so on. If the Listen Fixer Pointer has value 7 then the Listen Fixer is not written anywhere in the Listen Key. Note that Listen Key[115:112] cannot be overwritten by a Listen Fixer. This field is only used for TACL searches triggered by an LFLKC command.
TCP Source Port Upper Limit	This is the upper limit of a TCP source port that we will match against. It is inclusive, i.e. up to and including this value.
TCP Source Port Lower Limit	This is the lower limit of a TCP source port that we will match against. It is inclusive, i.e. greater than or equal to this value.
TCP Destination Port Upper Limit	This is the upper limit of a TCP destination port that we will match against. It is inclusive, i.e. up to and including this value.
TCP Destination Port Lower Limit	This is the lower limit of a TCP destination port that we will match against. It is inclusive, i.e. greater than or equal to this value.
IP Source Address	This is the 32-bit value that the incoming IP source address must match against after applying the IP Source Address Mask.
IP Destination Address	This is the 32-bit value that the incoming IP destination address must match against after applying the IP Destination Address Mask.
Listen Fixer	This is the 16-bit value that can be written into the Listen Key. See the Listen Fixer Pointer field above for more details. This field is only used for TACL searches triggered by an LFLKC command.

Table 5: TACL Entry Field Descriptions

2.5.3 Listen Fixers

As described in section 1.2.6.3, it is possible to modify the listen key based upon a matching TACL entry. This feature is useful only for the LFLKC command, where if the TACL search allows we perform a listen key search. To perform this feature the *Listen Fixer Pointer* and *Listen Fixer* fields of the TACL entry are used. The *Listen Fixer* field is a 16-bit value that can be written into the listen key. The *Listen Fixer Pointer* indicates where, if anywhere, the *Listen Fixer* should be written. This facility provides a mechanism to finely tune the listen entries that are matched against.

The expected usage of listen fixers is to overwrite portions of the listen key that are normally masked out to zero by the listen mask. We can then create listen entries that include these fixed values in the listen key. To create such listen entries, the LLKC command would be used. Note that the LLKC command does not apply any mask to the flow key when performing a comparison[23]. Instead it treats the full 116-bit flow key as a listen key, therefore allowing the LLKC command to create entries that have bits set outside of the listen mask.

One point to note about listen fixers is when the hash function is applied. For the LFLKC command, the listen key is formed from the flow key by applying a listen mask. The same hash function is then applied to the flow key and listen key. Based on these hash values, certain locks are applied for in the internal LUC architecture. Note that we do not know the final listen key value until after the *Listen Fixer* has been applied, so the hash that is computed, and the locks that were applied for, are based on the original unmodified listen key. When we do apply the *Listen Fixer* we do not re-compute the hash since that would mean we would have to modify our locks. Instead we search down the hash list looking for an exact match on the listen key, which now includes the *Listen Fixer* bits. Due to this implementation, the following point should be noted:

- All listen entries that have the same unmodified (pre *Listen Fixer*) listen key are on the same hash list. Therefore, values with the same unmodified listen key, but different modified (post *Listen Fixer*)

[23] It does apply the listen mask to the flow key when determining the listen hash value.

keys will be in a single hash chain. The length of this hash chain, and therefore the lookup performance, is directly proportional to the number of *Listen Fixer* values for the same listen key.

2.5.4 TACL Table Management

The only interface to the TACL table is via the DDR register interface described in section 4.8.24 on the DDR_ADDR register. It is the task of a management CPU to ensure that this table is maintained correctly. The following points should be noted:

1. Use the default TACL rule wherever possible, i.e. do not use a *match all* end rule.
2. When deleting a rule, the *Valid* bit can be used. This saves you having to copy the whole of the TACL table down one position. However, after some period of time the table should be completely re-written to remove the entries that are not marked as Valid. Leaving invalid entries in the TACL table can decrease performance.

It should be noted that while the TACL table is being updated, the LUC is still using it for searches. In fact, there is no way to stop the LUC from using the TACL table. For this reason the software must be very careful in the way it updates the TACL table. In the following sections we investigate the techniques for adding and deleting TACL entries.

2.5.4.1 Deleting the Last TACL Rule

To delete the last TACL rule:

1. Set the *Stop* bit in the last but one TACL rule.
2. Do not clear the *Stop* bit in the old last TACL rule, since the LUC may still be reading it. Instead leave the stop bit set until you need to overwrite it. Note that if you follow the steps below, when you do need to overwrite this rule you will overwrite it with a rule that has the *Stop* bit set.

2.5.4.2 Deleting a Non-Last TACL Rule

The simple way to do this is to clear the *Valid* bit for that rule. However, if you want to move all rules up by one then:

1. If you are deleting the N'th TACL rule then copy the N+1'th rule over the top of the N'th rule.
2. Delete the N+1'th rule, allowing for the fact that it might be the last rule.

2.5.4.3 Inserting a New Last TACL Rule

To Insert a new rule which is the last TACL rule:

1. If the last rule is in position N, then write the new last rule at position N+1. Make sure it has the *Stop* bit set.
2. Clear the *Stop* bit in the previous last TACL rule.

2.5.4.4 Inserting a New Non-Last TACL Rule

To insert a new rule that is not the last rule you need to make space in the TACL table. This is done as follows:

1. Copy the last rule, at position N, to position N+1.
2. Copy rule N-1 to position N.
3. Copy rule N-2 to position N-1.
4. Repeat until there is a space in the location where the new rule should be written.
5. Write the new rule.

2.6 *Timers*

In this section we examine the timer support that the LUC provides. We concentrate our discussion on TCP timers, but there is no reason why these timers cannot be used for other purposes when TCP is not the protocol being terminated.

2.6.1 Requirements

TCP maintains seven timers for each connection. These seven timers are (based on TCP/IP Illustrated Volume 2):

1. **Connection Establishment Timer:** Starts when a SYN is sent. If no response is received within 75 seconds then the connection is aborted.
2. **Retransmission Timer:** Used to re-send data that does not get acknowledged by the remote end. This timeout varies during the lifetime of a connection dependent upon the round trip time (RTT), but TCP limits its range from 1 to 64 seconds.
3. **Delayed ACK Timer:** When TCP receives data and wants to send an ACK, it sets the Delayed ACK Timer. This might allow it to acknowledge multiple receives frames. This timer has a 200ms resolution.
4. **Persist Timer:** This timer is set when the remote machine advertises a window of zero. This allows the local TCP to probe the window just incase window advertisements get lost. As with the retransmission timer, this timer varies during the lifetime of a connection dependent upon the RTT, but TCP limits its range from 5 to 60 seconds.
5. **Keepallve Timer:** This timer allows us to check that a connection is still active. Every two hours a special segment is sent to the remote machine. If it responds correctly then nothing is done and the keepalive timer is restarted, otherwise the connection is reset.
6. **FIN_WAIT_2 Timer:** When an application closes a connection we enter the FIN_WAIT_1 state and send a FIN. When that FIN is acknowledged we enter the FIN_WAIT_2 state: we are effectively waiting for the remote machine to send us a FIN. It could be possible that this FIN gets dropped, or the remote machine does not send it, so we use the FIN_WAIT_2 timer. It is set to 10 minutes. After this 10 minute timer has expired it checks the connection every 75 seconds, and will close it if the connection is idle.
7. **2MSL Timer:** This is called the 2MSL timer since the period is twice the maximum segment lifetime. When an application closes a connection it enters the FIN_WAIT_1 state and sends a FIN. When that FIN is acknowledged and the remote machine also sends a FIN we enter the TIME_WAIT state. This timer is set when we enter the TIME_WAIT state, and has a period of 1 minute.

In a traditional BSD TCP/IP stack all these timers are ticked every 500ms, except for the delayed ACK timer that has a resolution of 200ms. It should also be noted that the delayed ACK timer is a single shot timer: setting the delayed ACK timer means we would like to be timed out in the next 200ms tick.

Two pairs of these timers are also mutually exclusive, i.e. only one timer can exist at a time. The first pair is the connection establishment timer and the keepalive timer: the connection establishment timer is only active *while* the connection is being established, and the keepalive timer is only active *after* the connection has been established. Similarly, the FIN_WAIT_2 timer and 2MSL timer are mutually exclusive: each timer is tied to a specific state, and we can only be in one state at a time. If we count the delayed ACK timer as a special case one-shot timer, then the remaining six timers can therefore be implemented using four real timers.

In addition to the protocol timers we must also support some application specific timers for use by the interface cores or the host CPU.

Table 6 illustrates the various timers that the LUC must support, and the range and resolution of these timers on a BSD machine. In the BSD software the timers are typically limited to the size of the 16 or 32-bit word, but in the LUC we must be more conservative on the use of memory. For that reason we may not be able to support the full range that software based timers can. However, the range of the LUC timers can be expanded by keeping a separate *timer expired* count in the workspace: we simply count multiple LUC timer expirations. This allows us to expand the timer range at the expense of extra work on the protocol cores. The goal is that for normal operation we are not required to count expirations.

A BSD TCP/IP stack uses a 500ms resolution for all timers other than the Delayed ACK timer for which it uses 200ms. In the LUC we use a resolution of 2 seconds on all but the Delayed ACK, Retransmission and Persist timers for which we use 200ms. The timers with this 2 second resolution tend to be 75 seconds to 2 hours in length, so the 2 second accuracy should be acceptable when compared to the 500ms accuracy of BSD.

We assume that for protocols other than TCP these timers are adequate.

ACP Timer Number	Name	BSD			LUC Timer Control Unit		
		Min.	Max.	Resolution	Min.	Max.	Resolution
Protocol Timer 1	Delayed ACK	Default is 200ms. Can be configured.		200ms	200ms	1s	200ms
~~Protocol Timer 2~~	~~Retransmission Timer~~	~~1s~~	~~64s~~	~~500ms~~	~~200ms~~	~~101.8s~~	~~200ms~~
Protocol Timer 3	Connection Establishment Timer	Default is 75s. Can be configured.		500ms	2s	2.2 hours	2s
	Keepalive Timer	Default is 2 hours. Can be configured.					
Protocol Timer 4	FIN_WAIT_2 Timer	Default is 10m + 75s. Can be configured.		500ms	2s	16.9m	2s
	2MSL Timer	Default is 1m. Can be configured.					
Protocol Timer 5	Persist Timer	5s	60s	500ms	200ms	101.8 s	200ms
Application Timer 1	Fast Application Timer	N/A	N/A	N/A	200ms	50.6s	200ms
Application Timer 2	Medium Application Timer	N/A	N/A	N/A	2s	122s	2s
Application Timer 3	Long Application Timer	N/A	N/A	N/A	2s	8.4m	2s

Table 6: TCP and Application Timers

2.6.2 Timer Table

To maintain timer information the LUC uses a timer table consisting of timer entries. Figure 12 illustrates the format of a B10 timer entry. Each B10 timer entry is 32-bytes[24], allowing it to record the timers for four flows. For the S10 each timer entry is 64-bytes, allowing it to record the timers for eight flows. The format of the S10 is exactly the same as the B10 except that the timers for flows four through seven are added at the bottom.

If the LUC can support N_{FLOWS} flows, then the timer table must record the timers for N_{FLOWS} flows. To find the timers for a flow is a simple operation:

- **B10**: simply take *Socket ID*[21:2] and use it as an index into the timer table to obtain the 32-byte timer entry. Then use *Socket ID*[1:0] to pick which set of timers to use.
- **S10**: simply take *Socket ID*[21:3] and use it as an index into the timer table to obtain the 64-byte timer entry. Then use *Socket ID*[2:0] to pick which set of timers to use.

[24] 32-bytes is the minimum unit the B10 LUC can access in DDR memory.

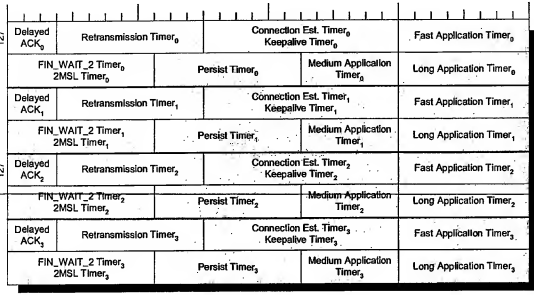

Figure 12: B10 Timer Entry Format

Table 7 defines the values of these timers. Note that as far as the software is concerned there are two values for an expired timer: all ones and all ones minus one. The software should check for both of these values[25].

Timer Value	Description
0	The timer is disabled and will not be ticked.
All ones	The timer has expired, and a CRTIMER command was successfully issued to the FDC.
All ones minus 1	The timer has expired, but the CRTIMER FDC command was either not successful or has not yet been issued.
Other values	These indicate the actual amount of time left for this timer. For example, if a 200ms timer has value 2 then it will expire in the next 200 to 400ms.

Table 7: Timer Values

2.7 *Free Block Management*

The LUC builds and maintains dynamic data structures. It therefore requires access to some kind of dynamic memory allocation, so that it can allocate such things as flow state entries. In this section we investigate the general mechanism that is used for block allocation, and then indicate the various available tables that use this mechanism.

2.7.1 Operation

For each set of blocks that the LUC needs to maintain free/busy information for, we create a cyclic buffer of indexes. At start of day this table is filled with an index to each block that is in the system. It should be noted that we maintain indexes, and not actual pointer values. For example, if we are storing available flow

[25] The software must process any timers that have value all ones. Failure to do this will result in timers becoming block for that flow, since the TCU will think that the timer has already been issued. The software *may* process any timers that have value all ones minus one. Any of those timers that are not processed will cause a timer expiration when the flow is checked in. Processing the expiration of the all ones minus one timers will save LUC resources.

states then each index refers to a single flow state. To turn that index into a DDR address you must multiply the index by the flow state size, and then add the base of the flow state table.

We maintain a read pointer, where indexes to free blocks can be obtained, and a write pointer where indexes to released blocks should be placed. Obviously the size of this cyclic buffer must be large enough to store an index to every block in the system: this is the case when **all** blocks are free. The number of indexes in this cyclic buffer, N_{FREE}, is used to determine when to wrap[26]. This data structure is the *DDR Cyclic Buffer* of Figure 13.

Since this table is held in DDR memory it would be expensive to access each and every time we need to obtain or release a block. We therefore use a caching mechanism so that the free/busy information for some smaller number of blocks is maintained in on-chip memory. The mechanism we use is a FIFO of free indexes with a pointer to the top of the FIFO. To obtain an index we simply read from the FIFO pointer, and decrement it. To put an index back on the free list we increment the FIFO pointer and then write the index to the free block. This is the *Free Block FIFO (On-Chip)* portion of Figure 13.

There are four watermarks maintained for the Free Block FIFO of Figure 13. They are used as follows:

1. **Write Watermark.** If the FIFO pointer goes above this watermark then a block of indexes is written out to the circular buffer in DDR memory. Note that we only trigger the write: when DDR memory is available we will actually perform the write.
2. **Read Watermark.** If the FIFO pointer goes below this watermark then a block of indexes is read in from DDR memory. Note that we only trigger a read from DDR memory: we do not wait for a response.
3. **Write Block Watermark.** If the FIFO pointer goes above this watermark then we are blocked from putting free indexes back into the FIFO (FIFO write). The reason is that we may have triggered a read from DDR memory, and we must always have a DDR bursts worth of indexes available to read into.
4. **Read Block Watermark.** If the FIFO pointer goes below this watermark then we must block any future requests for free indexes (FIFO read). The reason is that we may have triggered a write to DDR memory, so we must have at least a DDR bursts worth of indexes available to write.

[26] Note that N_{FREE} need not exactly match the amount of space reserved for the available table. See section 2.7.2 for more details.

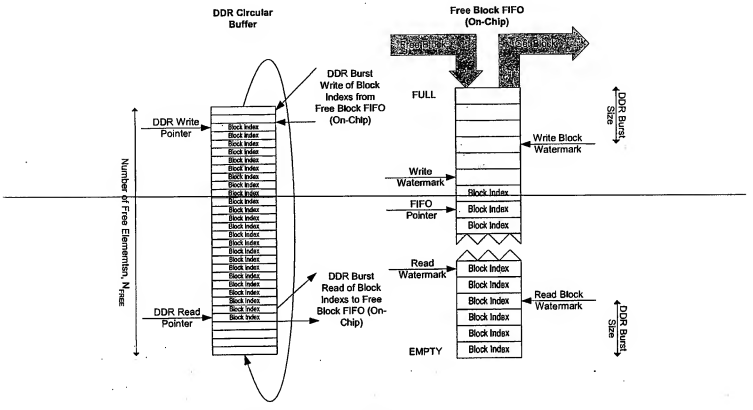

Figure 13: Free Block Management Data Structures

2.7.2 Initialisation

At boot time, the various Available Tables in the LUC must be initialised with valid *Block Indexes*. To do this the LUC must be configured with the base of the available table, and the number of elements, N_{FREE}, in that table. These values are supplied via the configuration registers. Note that it is not a requirement that N_{FREE} Block Indexes consume the whole space reserved for the Available Table, i.e. N_{FREE} can have any value, the only restriction being that it is a multiple of eight[27].

Given that the LUC knows the base and size of an Available Table, it must then initialise it with incrementing 32-bit Block Index values, starting at zero. The LUC does this using the FILLI and INIT bits of the CONTROL register. Once the base and size of every Available Table has been configured into the LUC registers, the CONTROL register should be written with FILLI set to one and INIT set to one. The LUC will then initialise all of the Available Tables in one go. See section 4.8.5 on the CONTROL register, and section 4.9 on initialisation for more details.

2.7.3 Available Tables

There are three tables that use the data structures described in section 2.7:

1. **Overflow Hash Entry AVTAB (Available Table).** This AVTAB maintains pointers to available entries in the Overflow Hash Entry Table. See section 2.3.1.2 for more information on hash table overflow.
2. **Flow State Entry AVTAB.** Maintains pointers to the available flow state entries of section 2.3.5.
3. **Listen State Entry AVTAB.** Maintains pointers to the available listen state entries of section 2.3.6.

[27] This is so that it fits in an S10 DDR burst.

2.8 LUC Resource Usage

2.8.1 LUC Parameters

There are a number of parameters in the LUC whose values must either must in a specific range, or must be one of an enumerated set of values. Table 8 defines those parameters.

LUC Parameter	Restrictions
Number of Hash Table Entries	Enumerated: 128K, 256K, 512K, 1M, 2M, 4M, 8M or 16M.
Number of Overflow Hash Table Entries	Range: Minimum 0, Maximum 4M, increments of 8.
Maximum Hash List Length	Fixed: 16K.
Flow State Size	Enumerated: 128 bytes, 256 bytes, 512 bytes, 1K bytes or 2K bytes.
Listen State Size	Enumerated: 128 bytes, 256 bytes, 512 bytes or 1K bytes.
~~Protocol / Interface Core Split~~	~~On any 128-bit boundary. Due to the granularity of the *Flow State Write Bitmap*, this is effectively limited to a 64-byte boundary. However, if one core uses the area as read-only then it may be possible to use a 128-bit boundary.~~
Number of Flows, N_{FLOWS}	Range: Minimum 0, Maximum 4M, increments of 8.
Number of Listen Entries, N_{LISTEN}	Range: Minimum 0, Maximum 512K, increments of 8.
Number of TACL Rules, N_{TACL}	Range: Minimum 0, Maximum 64K, increments of 1.

Table 8: LUC Parameters

2.8.2 LUC Memory Regions

The LUC maintains many tables in its DDR memory. This DDR memory is also configurable in size, allowing customers to reduce the memory requirements if only a small number of flows are required. In this section of the document we examine what tables are required and how their sizes are configured.

As stated previously, the S10 LUC has a dedicated 128-bit wide interface for up to 4GB of memory dedicated for flow state. The B10 LUC has a shared 64-bit wide interface for up to 16GB of memory that it shares with the SMC. For the B10, the LUC can address up to 4GB of DDR memory[28], but this 4GB of memory **must** start at physical address zero.

The LUC memory is divided into a number of regions, with each region storing a table. The start address of these regions is configurable, as defined in the registers of sections 4.8.11 through 4.8.18, subject to the rule that the B10 LUC resides in the first 4GB. These registers define the base of a table in terms of a 64KB pointer, therefore limiting the table sizes to multiples of 64KB. Before the LUC accesses a table it always adds the base address of the table to the table index to form the full DDR memory address.

Through out the following sections we use N_{FLOWS} to refer to the maximum number of flows the LUC has been configured for, and N_{LISTEN} to refer to the maximum number of listen entries.

2.8.2.1 Hash Table

The hash table takes a considerable chunk of the DDR memory. A good size for this table is to have four times as many hash entries as there are flows[29], i.e. the hash table should contain $4*N_{FLOWS}$ entries. This will help reduce the length of the hash lists so that the vast majority have length one. The longer the average length of a hash list, the longer it will take to find a match on the flow key. If performance is not an issue then this table could contain $2* N_{FLOWS}$ or even N_{FLOWS} hash entries: this will reduce the memory requirements considerably, at the expense of possible increasing the average hash list length.

[28] The LUC can address the whole of the 16GB of memory for debug access via its MMC registers. The 4GB limit is for hash tables, flow states etc.
[29] This is a rule of thumb.

2.8.2.2 Overflow Hash Table

As defined in section 2.3.1.2, if there is a collision in the hash table then we link in a new hash entry. The Overflow Hash Table is the area of LUC memory that contains these overflow hash entries. In the worst case there must be (N_{FLOWS} – 1) overflow hash entries: this allows for all entries hashing to the same value. However, if this is the case then performance will be severely degraded. With a reasonable hash function this worst case should happen with extremely low probability.

A more reasonable rule would be to say that the overflow hash table should contain N_{FLOWS}/2 hash entries. This would allow for every hash lists to be of length two. However, if the hash function is assumed to be good, then this table could be reduced even further.

If the overflow hash table becomes full then any future requests for overflow hash entries will be denied. In this case the protocol core will receive a workspace from the LUC, but the response field will say, "Entry could not be created due to lack of space".

2.8.2.3 Available Overflow Hash Entry Table

As described in section 2.7, for each dynamic element that can be allocated we must maintain an available table. The available overflow hash entry table maintains indexes to the overflow hash entries that are not currently assigned. We must allow for all overflow hash entries being available, i.e. the size of the available overflow hash entry table must be large enough to contain an index for each overflow hash table entry of section 2.8.2.2.

2.8.2.4 Timer Table

The timer table must contain enough timers for the maximum number of configured flows, N_{FLOWS}.

2.8.2.5 Flow State Table

The flow state table must contain enough space for the maximum number of configured flows, N_{FLOWS}.

2.8.2.6 Available Flow State Table

The available flow state table must be large enough to maintain an index to all flow state entries of section 2.8.2.5.

2.8.2.7 Listen State Table

The listen state table must contain enough space for the maximum number of configured listen entries, N_{LISTEN}.

2.8.2.8 Available Listen State Table

The available listen state table must be large enough to maintain an index to all listen state entries of section 2.8.2.7.

2.8.2.9 Termination Access Control List Table

The termination access control list table stores the rules that are used when access control is turned on. The size of this table is directly proportional to the maximum number of access control rules that have been configured into the LUC. If TACL functionality is not enabled then this table need not be configured.

2.8.3 Configuration Rules

Table 9 defines the rules for the various LUC memory regions described in section2.8.2. All but the hash table and overflow hash table are governed by a fixed equation.

All size values in Table 9 are in units of bytes. $S_{POINTER}$ is the size of a pointer in LUC memory, which is 4-bytes. S_{FLOW} and S_{LISTEN} are the sizes of a flow state and listen state accordingly, both measured bytes[30]. N_{FLOWS} represents the configured maximum number of flows, N_{LISTEN} is the configured maximum number of listen entries and N_{TACL} is the configured maximum number of TACL rules. S_{DDR_BURST} is the DDR burst size. This is 32-bytes for the B10 and 64-bytes for the S10. Note that since these tables are defined in terms of 64KB blocks, the minimum size table is 64KB.

Table	Size
Hash Table	Variable from 128K to 16M entries: A good size is $S_{HT} = 4 * N_{FLOWS} * S_{DDR\_BURST}$.
Overflow Hash Table	Variable: A good size is $S_{OHT} = N_{FLOWS} / 2 * S_{DDR\_BURST}$. Maximum is 256MB (S10) or 128MB (B10) due to the 22-bit Next Pointer in the Hash Entry (Figure 7).
Available Overflow Hash Table	Fixed by number of overflow hash entries: $S_{AOHT} = N_{OHT} * S_{POINTER}$.
Timer Table	Fixed by number of flows: $S_{TT} = N_{FLOWS} * 8$.
Flow State Table	Fixed by the number of flows: $S_{FST} = S_{FLOW} * N_{FLOWS}$.
Available Flow State Table	Fixed by the number of flows: $S_{AFST} = N_{FLOWS} * S_{POINTER}$.
Listen State Table	Fixed by the number of listen entries: $S_{LST} = S_{LISTEN} * N_{LISTEN}$.
Available Listen State Table	Fixed by the number of listen entries: $S_{ALST} = N_{LISTEN} * S_{POINTER}$.
Termination Access Control List Table	Fixed by the number of termination access control rules: $S_{TACLT} = N_{TACL} * 32$.

Table 9: LUC Memory Configuration Rules

2.8.4 Example Configurations

In the following sections we provide three example memory configurations using the rules of Table 9. The configurations provide both the sizes of the tables, and the settings of the registers in sections 4.8.11 through 4.8.18.

2.8.4.1 2GB of Memory, Maximize Performance

In this section we assume that 2GB of DDR memory are available to the LUC and that we wish to maximize performance, i.e. we want to try and keep the length of the hash lists small. This scenario could be used for a layer 7 switch. Note that 4GB is the maximum amount of memory that the LUC can address.

Table 10 defines the parameters we are going to use for this memory configuration. We use a large hash table compared to the number of flows, and allow for a reasonable number of overflow hash table entries. Since the hash table is large we should have small hash lists.

Parameter	Value	Register Settings
Number of flows, N_{FLOWS}	2,097,152 (2M)	FLOW_CNT.CNT = 2,097,152 / 8
Hash table size in entries	$4 * N_{FLOWS}$	HASH_PARAMS.K_FOLD = 2
Size of a flow state, S_{FLOW}	512 bytes.	LUC_PARAMS.FLOW_SIZE = 3'b010
Overflow hash table size in entries	$N_{FLOWS} / 2$	OVFLOW_CNT.CNT = 2,097,152 / 2 / 8
Number of listen entries, N_{LISTEN}	16,384 (16K)	LISTEN_CNT.CNT = 16,384 / 8
Size of a listen state, S_{LISTEN}	128 bytes	LUC_PARAMS.LISTEN_SIZE = 2'b00
Number of TACL rules, N_{TACL}	0	N/A

Table 10: Parameters for 2GB of Memory, Maximize Performance

Table 11 defines the table sizes and register settings for the *2GB of Memory, Maximize Performance* scenario.

[30] To ease the memory management task, S_{FLOW} and S_{LISTEN} must be powers of two.

Table	B10 Size (Bytes)	S10 Size (Bytes)	B10 Register Settings (Hex)
Hash Table	256M	512M	HT_OHT_BASE.HT_BASE = 0000
Overflow Hash Table	32M	64M	HT_OHT_BASE.OHT_BASE = 1000
Available Overflow Hash Table	4M	4M	AOHT_ALST_BASE.AOHT_BASE = 1200
Timer Table	16M	16M	TACL_TMT_BASE.TMT_BASE = 1240
Flow State Table	1G	1G	FST_LST_BASE.FST_BASE = 1340
Available Flow State Table	8M	8M	AFST_BASE.AFST_BASE = 5340
Listen State Table	2M	2M	FST_LST_BASE.LST_BASE = 53C0
Available Listen State Table	64K	64K	AOHT_ALST_BASE.ALST_BASE = 53E0
Termination Access Control List Table	0	0	N/A
Total	1,342M	1,626M	

Table 11: Table Sizes, 2GB of Memory, Maximize Performance

A first glance at Table 11 would seem to suggest that for the B10 we could increase the size of the flow from 512 bytes to 862 bytes. That would then use up the approximately 700MB of memory that is available for the B10. However, for memory management reasons the flow state size must be a power of two. Instead we could use this memory for socket page buffers, increased overflow hash entries or extra flows.

2.8.4.2 2GB of Memory, Maximize Number of Flows

In this section we assume that 2GB of DDR memory are available to the LUC and that we wish to maximize the number of flows, i.e. we are willing to keep the hash table smaller so that we can have a larger number of flows. Using a smaller hash table might cause longer lookup times, which would decrease performance. This configuration could be used for a layer 7 switch where the number of flows is more important then full performance.

Table 12 defines the parameters we are going to use for this memory configuration. In this configuration we are going to maintain 3M flows. The hash table must be a power of two, so rather than use 4 * N_{FLOWS} we use 4 / 3 * N_{FLOWS}, which is a smaller hash table. This is where we are decreasing the size of the hash table in order to allow for larger numbers of flows. Note that we still use the suggested overflow hash table size of N_{FLOWS} / 2.

Parameter	Value	Register Settings
Number of flows, N_{FLOWS}	3,145,728 (3M)	FLOW_CNT.CNT = 3,145,728 / 8
Hash table size in entries	4 / 3 * N_{FLOWS}	HASH_PARAMS.K_FOLD = 2
Size of a flow state, S_{FLOW}	512 bytes	LUC_PARAMS.FLOW_SIZE = 3'b010
Overflow hash table size in entries	N_{FLOWS} * 2	OVFLOW_CNT.CNT = 3,145,728 / 3 / 8
Number of listen entries, N_{LISTEN}	16,384 (16K)	LISTEN_CNT.CNT = 16,384 / 8
Size of a listen state, S_{LISTEN}	128 bytes	LUC_PARAMS.LISTEN_SIZE = 2'b00
Number of TACL rules, N_{TACL}	0	N/A

Table 12: Parameters for 2GB of Memory, Maximize Number of Flows

Table 13 defines the table sizes and register settings for the *2GB of Memory, Maximize Number of Flows* scenario.

Table	B10 Size (Bytes)	S10 Size (Bytes)	B10 Register Settings (Hex)
Hash Table	128M	256M	HT_OHT_BASE.HT_BASE = 0000
Overflow Hash Table	48M	96M	HT_OHT_BASE.OHT_BASE = 0800
Available Overflow Hash Table	6M	6M	AOHT_ALST_BASE.AOHT_BASE = 0B00
Timer Table	24M	24M	TACL_TMT_BASE.TMT_BASE = 0B60
Flow State Table	1.5G	1.5G	FST_LST_BASE.FST_BASE = 0CE0
Available Flow State Table	12M	12M	AFST_BASE.AFST_BASE = 6CE0
Listen State Table	2M	2M	FST_LST_BASE.LST_BASE = 6DA0
Available Listen State Table	64K	64K	AOHT_ALST_BASE.ALST_BASE = 6DC0
Termination Access Control List Table	0	0	N/A
Total	1,756M	1,932M	

~~Table 13: Table Sizes, 2GB of Memory, Maximize Number of Flows~~

2.8.4.3 64MB bytes of Memory, Maximize Performance

In this section we assume that 64MB of DDR memory are available to the LUC and that we wish to maximize performance, i.e. we use hash table sizes that minimize the length of the hash lists. This configuration could be used for an IP storage device where the number of flows is small and memory costs are important.

Table 14 defines the parameters we are going to use for this memory configuration. We use a large hash table compared to the number of flows, and allow for a reasonable number of overflow hash table entries. Since the hash table is large we should have small hash lists.

Parameter	Value	Register Settings
Number of flows, N_{FLOWS}	65,536 (64K)	FLOW_CNT.CNT = 65,536 / 8
Hash table size in entries	4 * N_{FLOWS}	HASH_PARAMS.K_FOLD = 6
Size of a flow state, S_{FLOW}	512 bytes	LUC_PARAMS.FLOW_SIZE = 3'b010
Overflow hash table size in entries	N_{FLOWS} / 2	OVFLOW_CNT.CNT = 65,536 / 2 / 8
Number of listen entries, N_{LISTEN}	16,384 (16K)	LISTEN_CNT.CNT = 16,384 / 8
Size of a listen state, S_{LISTEN}	128 bytes	LUC_PARAMS.LISTEN_SIZE = 2'b00
Number of TACL rules, N_{TACL}	0	N/A

Table 14: Parameters for 64MB of Memory, Maximize Performance

Table 15 defines the table sizes and register settings for the *64MB of Memory, Maximize Performance* scenario.

Table	B10 Size (Bytes)	S10 Size (Bytes)	B10 Register Settings (Hex)
Hash Table	8M	16M	HT_OHT_BASE.HT_BASE = 0000
Overflow Hash Table	1M	2M	HT_OHT_BASE.OHT_BASE = 0080
Available Overflow Hash Table	128K	128K	AOHT_ALST_BASE.AOHT_BASE = 0090
Timer Table	512K	512K	TACL_TMT_BASE.TMT_BASE = 0092
Flow State Table	32M	32M	FST_LST_BASE.FST_BASE = 009A
Available Flow State Table	256K	256K	AFST_BASE.AFST_BASE = 029A
Listen State Table	2M	2M	FST_LST_BASE.LST_BASE = 029E
Available Listen State Table	64K	64K	AOHT_ALST_BASE.ALST_BASE = 02BE
Termination Access Control List Table	0	0	N/A
Total	44M	53M	

Table 15: Table Sizes, 64MB of Memory, Maximize Performance

2.8.4.4 Other Configurations

In the previous sections we have concentrated on scenarios where the flow state size is 512 bytes. If the flow state size were smaller then a larger number of flows could be supported. Obviously, if the flow state size is larger than 512 bytes then the total number of supported flows is reduced.

We also went to two extremes as far as DDR memory sizes were concerned: 2GB and 64MB. There are obviously configurations in between these two that support varying numbers of flows. There is also the configuration that can use the full 4GB of memory.

2.9 Random Number Management

TCP processing requires a good source of random numbers. This is an even stronger requirement when cryptography is involved, since the quality of the random numbers can directly affect the quality of the cryptography. In this section we discuss how the LUC can interface to an external random bit generator and provide the protocol / interface cores with a random seed sequence.

It should be noted that using an external random bit generator is not a requirement. In some cases the host can provide a seed to the protocol cores via a host event. The protocol cores could then use this seed in a pseudo-random number algorithm. If the host also has a random bit generator then it could repeatedly re-seed the protocol cores using host events.

It would also be possible to include a random bit generator on the ASIC itself. However, doing this would require us to prove that it truly is a random bit sequence, which can be quite a task for cryptographic applications.

2.9.1 External Random Bit Generators

To provide a good source of random numbers, the LUC can use an external random bit generation (RBG) device. Such a device is the RBG-1210 from Tundra Semiconductors. This device uses two signals to communicate random bits: a strobe and a data. Using this device you can get a random bit sequence at about 20KHz. The RBG-1210 will be the target device for the LUC to interface to, although whether such a device is available is a configurable option via the EN_RBG bit of the CONTROL register.

2.9.2 Random Number Collection

The LUC must gather the bits from the RBG and put them into 16-bit registers. There are ten such registers for the B10, and fifteen for the S10: one for each of the Processor Cores. Each register also has a *fresh* bit. This bit is set when the 16-bit register has been filled with random bits, and is cleared when the value of the register is read.

The LUC maintains a pointer into this array of registers, and each time a full 16-bits is filled it increments its pointer. The LUC continually maintains this operation. If a random register for a core is not read before a new random value is available, then the previous random value is simply overwritten.

2.9.3 Random Number Distribution

We now have a set of 16-bit registers and a set of fresh bits. If the fresh bit is set then it means we have a new random value for a core. These random values are distributed to the cores whenever a workspace is sent. For this purpose we use the RSV and Random Seed fields of the workspace format, as defined in section 4.5.1. The operation is as follows:

1. If the fresh bit is set then set the RSV bit of the workspace and read the 16-bit random register into the random seed field. The read of this random register will clear the fresh bit.
2. If the fresh bit is not set then clear the RSV bit of the workspace and set the random seed to value zero.

Using this mechanism the cores can get access to a random number sequence. Since we only get a random bit approximately every 50us, we can only supply a core with a 16-bit random quantity approximately every 16ms. This is neither enough bits nor often enough for the core to use this as a random sequence directly.

Instead it should collect these 16-bit random values into 32-bit seeds. It can then use a local pseudo-random number sequence algorithm that it should re-seed every time it has a full 32-bit random value collected from the various workspaces it receives.

Note: if the random bit generator is not configured for a customer then the Processor Cores must seed their random number algorithms using some other method. However, we do expect this to lead to a lower quality random number sequence when compared to using a RBG.

2.10 Global Timer Support

The ACP consists of multiple processors that are operating in a parallel fashion. Some of the tasks that these processors are performing involve the concept of time passed. For example, an interesting metric is to measure the time between when a SYN was sent to a server, and when that server's application responded with its first data packet. Another example is the Timestamp option of TCP, used for Round Trip Time (RTT) measurements. The ACP architecture is such that different Processor Cores may process these two events. The Processor Cores storing a time stamp in the flow state can solve this. The problem here is what clock do the Processor Cores use that is synchronised between all Processor Cores? This is where LUC support for a global timer is used.

2.10.1 Operation

The LUC exports two signals to all Protocol Clusters. These signals are the GLOBAL_TIMER_CLK and GLOBAL_TIMER_CLR signals, as illustrated in Figure 1. The Protocol Clusters use these signals to control the operation of a 52-bit register that is accessible by the Processor Cores of that cluster. Table 16 defines the two signals that the LUC exports for global timer usage. These two signals allow the 52-bit count that is on the Protocol Cluster to be used as a global timer value by the Processor Cores.

Signal	Description	Related LUC Register
GLOBAL_TIMER_CLK	Each time this signal cycles, all Protocol Clusters increment their 52-bit counter.	CONFIG.GTM_CYCLE (section 4.8.6).
GLOBAL_TIMER_CLR	When this signal is cycled, all Protocol Clusters reset their 52-bit counter.	CONTROL.GSTAMP_CLR (section 4.8.5).

Table 16: Global Timer Signals

Note that these two signals must have equal amounts of registration between the LUC and all Protocol Clusters. If they do not then the increment and clear signals will be skewed.

2.10.2 Configuration

The LUC uses a 20-bit value to indicate how many Protocol Cluster clocks it should wait before cycling GLOBAL_TIMER_CLK. This value is the GTM_CYCLE field of the Configuration register (see section 4.8.6). The minimum value of this register is 10, therefore providing a minimum GLOBAL_TIMER_CLK cycle time of 37.594ns with a 266MHz Protocol Cluster clock. This gives a period of between 37.594ns and 3.942ms. Since the counter on the Protocol Cluster is 52-bits wide, this means it will wrap from between 5.4 and 500,000 years.

The GTM_CLR bit in the control register (see section 4.8.5) controls the GLOBAL_TIMER_CLR signal of the LUC. When a Processor Core writes a 1 to this field, the GLOBAL_TIMER_CLR signal is raised. Writing a 0 to this field lowers GLOBAL_TIMER_CLR. At the start of day this signal should be toggled to ensure that all Protocol Cluster counters are synchronised.

2.11 Error Correction and Detection

In order to protect the large amount of DDR memory that the LUC manages, the LUC implements Single Error Correction and Double Error Detection (SECDED) logic. The LUC will automatically detect and fix any single bit errors. If such a correction is made, then the LUC sets the SEC_ECC_ERR bit of the STATUS

register. The LUC will also detect any double bit errors, setting the DET_ECC_ERR bit of the STATUS register (section 4.8.4) if any are found.

In order to test this logic the LUC provides a mechanism where by bits can be flipped while they are being written to DDR (but after the ECC has been computed). This is done via a 72-bit flip mask. If during a write bit i is set in the flip mask, then bit i of the write data is flipped as it is written to DDR. 72-bits are used since this allows to you flip the 8-bits of ECC information. Note that this is only done if the DDR_DO_FLIPEN bit of the DEBUG register (section 4.8.26) is set.

The 72-bit flip mask is shifted into the LUC via successive writes of the DEBUG register. Each write of the DDR_DO_FLIP fields shifts the current 72-bit value left by 16, and then adds in DDR_DO_FLIP to the bottom.

3 LUC Functional Units

3.1 High Level LUC Block Diagram

Figure 14 illustrates the internal elements of the LUC. The LUC uses its own clock domain that matches that of the DDR memory[31].

The main blocks in Figure 14 are:

1. **LUC Engine Balancer.** This ensures that requests in the LUC Request Queue from the Dispatcher are distributed amongst multiple LUC Engines.
2. **LUC Engines.** These engines search through hash lists in DDR memory looking for a match on a flow or listen key. They also manage the hash lists.
3. **Timer Control Unit (TCU).** The LUC uses the Timer Control Unit (TCU) to decrement the counters in the timer table. This requires access to the DDR Memory as well as the Timer Cache, where the values of checked out timers are stored.
4. **LUC Workspace In Buffer.** This provides a buffering and re-assembly area for workspaces that are received back from the Processor Cores via the Message Bus.
5. **Flow Locker.** To ensure data integrity, each engine must first lock the hashes of the flow using the Flow Locker. This ensures that no two LUC Engines are working on the same hash lists.
6. **Timer Cache.** This provides a cache for the timers of a flow while that flow is checked out.
7. **DDR Memory Controller.** Provides access to the DDR memory for the Timer Control Unit and LUC Engines. For the B10 part the DDR memory controller is external to the LUC and is shared by the SMC.
8. **LUC Engine Combiner.** Merges the responses from the multiple LUC engines into the Check-Out buffer.
9. **Check-In Buffer and Check-Out Buffer.** These are 4KB dual port memories that allow for synchronisation between the Core and LUC clock domains. For workspaces between 128 and 512-bytes, each LUC Engine is assigned a fixed address in these buffers. For 1KB workspaces only four LUC Engines are used and the Check-In / Check-Out buffers are split into four. For 2KB workspaces only two LUC Engines are used and the Check-In / Check-Out buffers are split into two[32].

[31] At present the DDR memory and the majority of the internal logic use the same clock speed: 266MHz. However, in order to improve performance it may be feasible to use a faster DDR memory, therefore requiring that the LUC be in its own clock domain.

[32] Even though there are only two LUC engines operating, the DDR bus utilisation should still be very good due to the very large blocks that are being read / written (2KB).

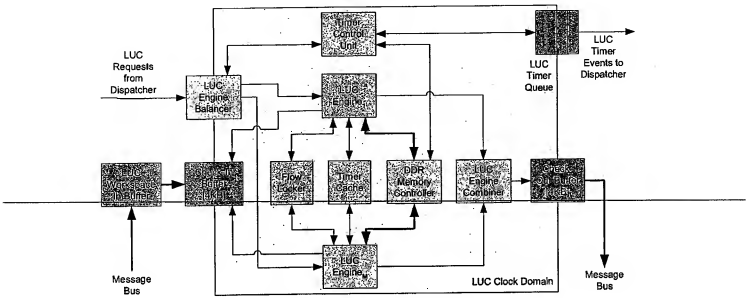

Figure 14: High Level LUC Block Diagram

The details of the LUC Engine Balancer, DDR Memory Controller and LUC Engine Combiner are beyond the scope of this document. They are included in the above diagram for completeness. In the following sections we define the operation of the other blocks in the LUC.

3.2 Flow Locker

As previously described, the LUC uses hashing to implement the lookup of the flow key. The LUC also maintains these hash lists, adding and deleting entries when requested to do so. Since multiple engines can access this single hashed data structure, we need a way to co-ordinate updates. For example, what would happen if we simultaneously tried to remove two elements from the same linked list; without proper locking the results will be undefined. This is where the flow locker is used.

3.2.1 Requirements

By examining the hashing algorithms, it can be seen that each hash list is independent of the others, i.e. we can safely operate on two or more hash lists without requiring any form of locking[33]. Therefore, if each engine were operating on a different hash list then we require no further locking, i.e. if each engine were working on a different hash value. The Flow Locker is therefore a CAM of flow hash values. Each engine, before it executes a command, inserts an entry in the flow locker. When the engine is finished that flow locker entry is removed.

As we will show later in this document, the operation of the LUC requires that two hash locations be locked before an engine can continue: one lock for the flow key, another lock for the listen key. In fact, this will be quite a common lookup. Note also that for lots of flows the listen key will be the same: it is commonly the destination IP address with the destination port. Due to this fact we cannot implement single locks, i.e. we cannot implement a scheme where only one LUC engine can have a lock at a single time. The reason is that if we did then it will be quite common to have all LUC engines stalled since they are all trying to claim the same listen hash value lock. We must perform a more complex form of locking to overcome this problem.

[33] This is ignoring shared resources such as a list of free hash locations etc. We assume that access to these resources is through some other locking mechanism than the flow locker.

When a flow hash value and listen hash value are locked, only one of these hash lists will potentially be modified, the other hash list being a read lock. The Flow Locker must therefore have the ability to atomically award read locks for a single hash value, write locks for a single hash value, or a ganged read lock and write lock for two hash values. The LUC should allow multiple read locks to be present on a single hash value, but only one write lock. A read and write lock can never exist on the same hash value. Given this locking scheme, it would be possible for multiple engines to have a read lock on the same listen hash value.

When the flow locker is presented with a ganged write lock and read lock, it must allow both locks or deny both locks, i.e. it cannot claim only one lock. We must also allow for the case where a stream of read locks could potentially indefinitely block out a single write lock: the write lock must get its fair share of locks. With these two rules we can ensure that the flow locker does not become deadlocked.

The size of the flow locker must allow for each engine to have a ganged write and read lock, i.e. the flow locker must have twice as many entries as there are LUC engines.

3.2.2 Operation

In this section we present the operation of the flow locker. This is for insight only, since the above requirements section provides enough detail for a designer to implement the feature.

Table 17 illustrates the format of an entry in the flow locker. Note that for the purposes of this document we do not spell out the exact bit locations for each of these fields. This is not important since the only device accessing this table is the LUC itself.

Field	Size	Description
Valid Bit	1-bit	Indicates if the entry in the flow hash locker is valid.
Hash Value	24-bits	This is the 24-bit hash value that is used as the key in the Flow Locker.
Claim	1-bit	If this bit is set to one then a write lock is pending on this hash value.
Write	1-bit	If this bit is set to one then a write lock is active on this hash value.
Read Count	4-bits	Count of the number of read locks that are active on this hash value, minus one.

Table 17: Flow Locker Entry

Table 18 illustrates the commands that the LUC will issue to the flow locker. One important point to note from this table is that in the ganged write lock with read lock, if either lock is denied then both locks are denied.

Table 18 also illustrates the use of the claim bit. This bit is set when a write lock is denied due to a read lock being present. If the bit is set then any future read locks are denied, effectively allowing the read locks to be removed. Once all read locks are removed, the write lock will be granted, clearing the claim bit. This meets our requirement of giving write locks their fair share of locking. Note that the ganged lock ignores the claim bit. This is required for specific scenarios that are explained later in section 3.2.3.

Command	Description
Read Lock	Deny if any write locks are present or pending, or if the claim bit is set. Allow otherwise.
Read Lock No Yield	Deny if any write locks are present. Allow otherwise. This ignores the claim bit.
Read Unlock	If the read count is zero then clear the valid bit, otherwise decrement the read count.
Write Lock	Deny if any write or read locks are present, allow otherwise. If a write lock is already present then set the claim bit. If the write lock is allowed then clear the claim bit.
Write Unlock	Clear the valid bit.
Ganged No Yield: Write Lock (FH) and Read Lock No Yield (LH)	Deny if write lock present on FH or LH. Deny if read lock present on FH and set the claim bit. Note that we ignore the claim bit of LH. If either the Write Lock or Read Lock No Yield is denied then both must be denied.
Ganged Unlock: Write Unlock (FH), Read Unlock (LH)	Clear the valid bit for FH. If the read count is zero for LH then clear the valid bit, otherwise decrement the read count.

Table 18: Flow Locker Commands

3.2.3 Special Case Scenarios

3.2.3.1 Flow Hash Equals Listen Hash

There is a special case of the ganged lock when the flow hash and listen hash are equal. In this case the ganged lock is transformed into a write lock, and the ganged unlock into a write unlock.

3.2.3.2 Two LFLKC Commands with Swapped Hash Values

Consider the following sequence:

1. LFLKC command arrives with Hash(Flow Key) = A, Hash(Listen Key) = B. Lets denote this by LFLKC[A,B].
2. LFLKC command arrives with Hash(Flow Key) = B, Hash(Listen Key) = A. Lets denote this by LFLKC[B,A].

Suppose the first LFLKC[A,B] is denied, perhaps because a write lock is already present on B. In this case the claim bit will be set for B. Now suppose that the LFKLKC[B,A] is denied, perhaps because a write lock is present on A. In this case the claim bit will be set for A. If we strictly adhered to the claim bits then we are now in a deadlock scenario, since neither LFLKC[A,B] or LFLKC[B,A] will succeed since they have both set each others claim bits. This deadlock will continue to exist even when the write locks of A and B are released.

To solve the above deadlock, the LFLKC[A,B] flow locker command uses the *Ganged No Yield* variant of Table 18. This ganged lock ignores the claim bit when it attempts a read lock on B. This prevents the deadlock scenario, but it introduces a starvation scenario described below.

3.2.3.3 Constant Stream of LFLKC Starves LLKC or LFKC

Given the explanation in section 3.2.3.2 above, a LFLKC[A,B] command (where A and B are hash values) ignores the claim bit when it attempts a read lock on B. This can introduce starvation problems. For example, consider the following sequence:

1. LFLKC[x, B] command arrives. Lets assume that it successfully gets its locks.
2. LLKC[B] or LFKC[B] command arrives. It fails since it wants a write lock on B, but the above command has a read lock on that hash value. The claim bit is set for B.
3. A constant stream of LFLKC[x, B] commands arrive. They continue to ignore the claim bit that was set in step 2 above.

With the above scenario, the LLKC (or LFKC) command will not be executed until the stream of LFLKC commands stops, i.e. the LLKC/LFKC commands are being starved. While this is a degenerate case, we must ensure that this condition cannot occur indefinitely. To fix this problem we expand the flow lock arbiter as follows:

1. Each time a lock request from a LUC Engine is denied, a counter for that engine is incremented.
2. Each time a lock request from a LUC Engine is granted, the counter is reset to zero.
3. If an Engines counter goes above a programmable value, then that engine is put in the highest priority pool. Lock requests from other engines in the low priority pool are denied (unlocks are allowed).

The HL_DENY_LIM value of the HASH_PARAMS register (section 4.8.23) defines the number of denies that can be received by a LUC Engine before it is put into the high priority pool. A value of 1024 is recommended since this should prevent the mechanism from interfering in the normal operation of the LUC, i.e. except for the degenerate case described above is should not be triggered. A value of zero disables this feature.

Note that this is not a deadlock issue, but is rather a starvation issue. For this reason is does not matter if multiple LUC Engines enter the high priority pool. Each of those engines will eventually be serviced, and they should all be serviced before any of them can re-enter that pool. Note that if the HL_DENY_LIM value is set too low then a LUC Engine could re-enter the high priority pool before all members of that pool have been serviced. This could again cause starvation. The recommended value of 1024 will not cause this to occur though.

3.3 Timer Cache

As described in section 2.3.6, the LUC keeps a number of timers per flow. One of the requirements of the LUC is that when a flow is checked out[34], no timer events can be issued for that flow. However, when a flow is checked out we must still decrement the timers for that flow: we simply cannot issue timer expired events.

The unit of access to the DDR memory on the S10 LUC is 64-bytes, and 32-bytes for the B10. Since the timer table is kept in DDR memory, and since each flow only uses 64-bits worth of timers, it will be the case that timers from multiple flows will be present in a single 64 or 32-byte value. This introduces read / modify / write problems. For example, consider the case when a flow has been checked out and is now being checked back in, i.e. the LUC has received an update command. The new timers for this flow must now be merged in with the existing timers in the 64 or 32-byte word. That read / modify / write operation on DDR memory is expensive, and should be avoided. It also introduces other problems where two engines may be modifying timers for different flows but the flows are in the same 64 or 32-byte word. To solve these problems the LUC uses a timer cache.

The S10 timer cache is a cache of 64-bytes worth of timers, i.e. eight flows worth of timers. For the B10 LUC this timer cache is 32-bytes worth of timers, i.e. four flows worth of timers. When a flow is checked out the DDR word containing the timers for that flow is brought into the timer cache. While that flow is checked out the timers are updated in the timer cache rather than the DDR memory. The timer cache is a CAM that is looked up using a shifted value of the Socket ID. Note also that when a timer tick occurs the LUC must first check the contents of the timer cache before retrieving an entry in the timer table.

Table 19 and Table 20 illustrate the format of an entry in the S10 and B10 Timer Cache. Note that for the purposes of this document we do not spell out the exact bit locations for each of these fields. This is not important since the only device accessing this table is the LUC itself.

[34] In the strictest sense, a flow is checked out when an entry for it exists in the FDC. As far as the LUC is concerned a check out is between the time the lookup request is received, and the time the update or teardown is received.

Field	Size	Description
Valid Bit	1-bit	Indicates if the entry in the timer cache is valid.
Lock Bit	1-bit	Indicates if this entry is currently being worked on and is locked.
Timer Cache Key	19-bits	This is the key for the timer cache. It is Socket ID[21:3]. Since each timer cache entry contains eight flows worth of timers, we shift the socket ID right by 3 to form the tag.
Checkout Bitmap	8-bits	Each bit indicates whether the timers for this flow are checked out or not. When this bitmap reaches zero the cache line can be written back to DDR memory and emptied.
Timers for Socket ID[2:0]	8 x 64-bits	Timers for each flow. A flows worth of timers is 64-bits.

Table 19: S10 Timer Cache Entry

Field	Size	Description
Valid Bit	1-bit	Indicates if the entry in the timer cache is valid.
Lock Bit	1-bit	Indicates if this entry is currently being worked on and is locked.
Timer Cache Key	20-bits	This is the key for the timer cache. It is Socket ID[21:2]. Since each timer cache entry contains four flows worth of timers, we shift the socket ID right by 2 to form the tag.
Checkout Bitmap	4-bits	Each bit indicates whether the timers for this flow are checked out or not. When this bitmap reaches zero the cache line can be written back to DDR memory and emptied.
Timers for Socket ID[1:0]	4 x 64-bits	Timers for each flow. A flows worth of timers is 64-bits.

Table 20: B10 Timer Cache Entry

The timer cache provides three simple commands:

1. **Lock Unconditional**: Given a socket ID, this looks up and locks a line in the cache. If a timer cache entry is found, but the lock bit is set, then this command is not acknowledged. If a timer cache entry does not exist then one is created and locked. The response of this command indicates if a timer cache entry had to be created or not.
2. **Unlock**: This clears the lock bit.
3. **Unlock and Remove**: This clears the valid bit.

Once a timer cache entry line has been locked the checkout bitmap and timers can be modified.

Using these commands the LUC engines manage the timer cache. For example, when performing a command such as LFK (Lookup with Flow Key), the LUC engine will use a Lock Unconditional command on the socket ID. It will then update the checkout bitmap to indicate that this flow is now checked out. Finally an unlock command leaves the entry in the timer cache entry but clears the lock bit. Any future operations on the timers of this flow will be performed upon the timer values in the timer cache.

The reason for the lock bit is to ensure that multiple units are not working on a cache line at the same time. For example, we need to ensure that the timer control unit does not collide with any other LUC engines while it is updating the timers of an entry in the timer cache.

The size of the timer cache must allow for the maximum number of flows being checked out at any one time. The Flow Director CAM HLD states that it contains 96 entries for the B10 part, therefore only 96 flows can be checked out at any instant in time. The timer cache will therefore contain 97 entries: this allows for the maximum number of flows plus one more for use by the timer control unit of the LUC[35]. Similarly, the S10 Flow Director CAM will contain 144 entries, therefore requiring the S10 time cache to be 145 entries.

[35] We assume that worst case in that each checked out flow requires a different timer cache line.

3.4 LUC Workspace In Buffer

The LUC Workspace In Buffer provides a buffer space for the workspaces that are sent from the Processor Cores via the Message Bus. The reason such buffering is required is because the Processor Cores send a workspace to the LUC each time they have finished processing an event for that flow. If that is the last event for the flow, then a message is sent to the Dispatcher informing it to update the LUC. It is only then that the LUC receives a command to perform the update. In between that time the workspace must be buffered, as should any other workspaces that are written back during that period.

It is not sufficient for the LUC to buffer a single workspace from each Processor Core. Doing this would require backpressure on this single buffer, and also complicates the splitting of workspaces between a Protocol and Interface Core[36]. The LUC must therefore be able to buffer and track multiple workspaces from a single Processor Core. The simplest way to do this is to directly map the Processor Cores workspace region of memory into an equally sized memory on the LUC.

The method described below for buffering the workspaces from the message bus is relatively simple at the expense of memory. Other schemes exist that would allow us to reduce the memory requirements at the expense of complexity. See section 5.2 for further details on some of these schemes, and why they were not chosen.

3.4.1 Buffer Layout

Each Processor Core uses between 1KB and 8KB of memory for workspaces, with 2KB the most common scenario. This memory is split into sixteen equal sized parts, with each part referenced by a Workspace ID. The LUC uses the exact same scheme. For each Processor Core, the LUC mirrors this memory in the LUC Workspace In Buffer.

The B10 LUC, which has ten Processor Cores, uses a 26KB area of memory for the LUC Workspace In Buffer. This size allows for the following configurations, with many variations in between:

- Ten Protocol/Interface Cores each using 2KB. [Requires 20KB]
- Eight Protocol Cores each using 2KB, and two Interface Cores using 4 or 5KB [Requires 24KB to 26KB].
- Eight Protocol Cores each using 1.25KB, and two Interface Cores using 8KB [Requires 26KB][37].

The S10 LUC has fifteen Processor Cores, i.e. it has 50% more Processor Cores than the B10. The S10 therefore uses 50% more for the LUC Workspace In Buffer, putting its requirement at 39KB.

The LUC uses a programmable offset into the LUC Workspace In Buffer to determine where the buffer area is for a specific Processor Core. This programmable offset is configured into the LWIB_PARS registers (see section 4.8.63). Note how the Core Index is used to index into the LUC Workspace In Buffer, and not the Core ID that is supplied in the LUC commands and Workspaces. The reason for this is that a Core ID is the value of {Cluster Controller, Core Number}, which is non-contiguous. The Core Index is a contiguous encoding of the Core ID, therefore saving on registers. See section 2.1 for an explanation of the numbering schemes that are used in the LUC and throughout the ACP (Pegasus). If, during any LUC Workspace In Buffer transactions, an invalid Core ID is presented to the LUC then the OPC_ERR bit is set in the STATUS register.

Figure 15 illustrates the scheme for the LUC Workspace In Buffer.

[36] When workspaces are split between two Processor Cores, you can no longer guarantee the order in which they are written back. This introduces problems with a single workspace buffer per Processor Core implementation.

[37] In this scenario, the Protocol Cores have had to reduce their workspace usage to 1.25KB in order to allow the Interface Core to use the full 8KB workspace.

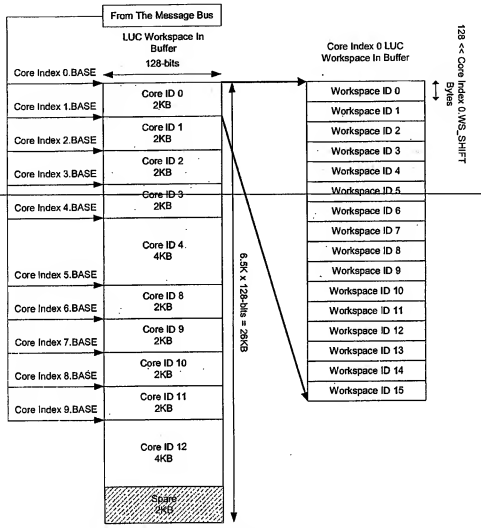

Figure 15: LUC Workspace In Buffer

In Figure 15 it can be seen how the BASE portion of the LWIB_PARS register is used to determine the base address for a given Processor Core. The exact area to use inside that base address is then given by the Workspace ID. Each region is split into sixteen Workspace IDs. Each Workspace ID is configurable in size from 128 to 512 bytes, using the WS_SHIFT parameter of the LWIB_PARS register. This WS_SHIFT parameter is exactly the same parameter that is programmed into the Protocol Cluster, and is used to allow a Processor Core to address between 2KB and 8KB of memory for workspace usage. See the *Protocol Cluster HLD* for further details.

Note that the Workspace ID is only used to determine the start of the Workspace – the actually workspace size may cause it to consume multiple Workspace ID locations. For example, if a Processor Core is using 2KB of memory for workspaces, and the workspace size is 512 bytes, then only Workspace IDs 0, 4, 8 and 12 are used. Each workspace starts at one of these workspace IDs, and then consumes the next four workspace IDs.

3.4.2 Tracking Buffer Readiness

As described in section 1.2.5, the LUC can operate in a mode where a flow state is split between two Processor Cores. Each Processor Core updates its own portion of the flow state. When both Processor Cores have finished, the LUC takes the two halves of the workspace, combines them, and writes them back as a single flow state.

In order to improve performance and simplify the design of applications, it is possible that only one Processor Core updates its portion of the flow state, and then it requests that the flow be updated. In this scenario the LUC will only receive one half of the workspace back. The other half is unchanged and the LUC must not modify it. In order to implement this feature the LUC must be able to track whether a Processor Core has written a specific workspace. It then uses this information to determine how to write back a flow state. This information is tracked in the READY bitmap. Note that either the Processor Core or the Interface Core has to write back the workspace (unless the flow is being torn down)[38].

The READY bitmap contains a single bit for each {Processor Core, Workspace ID} combination. For the B10, since there are ten Processor Cores each with sixteen Workspace IDs, this bitmap is 160-bits wide. On the S10 the number of Workspace IDs is the same, but there are fifteen Processor Cores, therefore requiring a 240-bit wide bitmap.

Each time a Processor Core writes to a Workspace ID, the READY bit is set to value one. The LUC can therefore use this bit to determine if a Processor Core has written to that Workspace ID. Just before the LUC sends a workspace to a Processor Core, it clears the READY bit. It is important that the ready bit is cleared at this point. Clearing the READY bit after each update could cause the READY bitmap to become inaccurate. The reason is that the Processor Cores are allowed to write back workspaces even while a USID command for that workspace is being processed[39]. Clearing the READY bitmap when a workspace is checked out avoids this issue.

The READY bitmap is indexed by the 8-bit value formed by concatenating the 4-bit Core Index with the 4-bit Workspace ID: {Core Index, Workspace ID}.

3.4.3 Message Bus Receive Flow Chart

The *Message Bus Receive* flow chart of Figure 16 illustrates the steps that the LUC should take when it receives a message from the Message Bus. This message will be in the format defined in Figure 42, the Workspace format. Table 21 defines the variables that are used in this flow chart.

Field	Description
core_index	The Protocol Core Index. This is an encoded value of the Protocol Core ID. See section 2.1 for more details.
core_base	This is the base address of where workspaces are stored for this Processor Core Index. This comes from the LWIB_PARS registers of section 4.8.63. For more information on the LUC Workspace In Buffer and its configuration the reader is directed to section 3.4.

[38] When exclusive flow state splitting (see section 2.3.5.3) is used, the Protocol Core **must** write back the first chunk of its workspace when a flow is created. When shared flow state splitting is used (see section 2.3.5.4), then either the Protocol Core or the Interface Core **must** write back the first chunk of the shared area when a flow is created. Failure to do this will result in incorrect timer expirations. See section 2.3.5.6 for the full set of rules regarding workspace write backs.

[39] This occurs when an event arrives at the Dispatcher just after a USID command was issued.

Field	Description
core_shift	A Protocol Cores area in the LUC Workspace In Buffer is split into sixteen different sections. The Workspace ID of a LUC command indicates which of these sections is being referenced. The size of each section is 128-bytes, 256-bytes or 512-bytes depending on the value of the WS_SHIFT field of the LWIB_PARS register for that Protocol Core. The core_shift variable contains the amount to shift the Workspace ID by to turn it into a 128-bit offset into the LUC Workspace In Buffer. The minimum workspace size (WS_SHIFT = 0) is 128-bytes, which requires a shift of 3 to turn a Workspace ID into a 128-bit offset. Workspaces of 256-bytes (WS_SHIFT = 1) and 512-bytes (WS_SHIFT = 2) are allowed for by adding 3 to the WS_SHIFT value.
core_addr	This is an address into the LUC Workspace In Buffer that points to the start of the workspace. It is calculated from core_base and core_shift. This is a pointer to a 128-bit word.

Table 21: Message Bus Receive Flow Chart Variables

The first task of Figure 16 is to wait for a message to arrive. When one does arrive we calculate the core_index[40], core_shift and core_addr variables according to Table 21. The core_addr variable is the goal of these calculations – we want to find out where to write this workspace in the LUC Workspace In Buffer.

After the entire message has been written into the LUC Workspace In Buffer we update the READY bitmap to indicate that this workspace ID is now valid. We then go back to the start and wait for another message bus transaction to arrive.

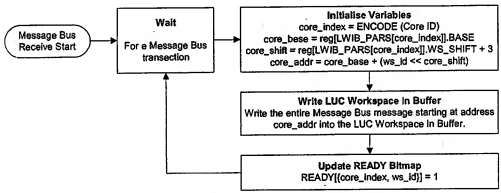

Figure 16: Message Bus Receive Flow Chart

3.5 LUC Engines

In the following sections we describe the processing that the LUC Engines perform. There are eight LUC Engines in total. For workspaces between 128 and 512-bytes all eight engines can process the same type of operation (lookup or update). However, for 1KB workspaces only four engines can process the same type of operation, and for 2KB workspaces only two engines[41]. This is done to limit the size of the Check-In and Check-Out buffers of Figure 14 to 4KB. See section 3.1 for more information on the Check-In and Check-Out buffers.

We describe the operation of the LUC Engines as follows:

- First we discuss the two subroutines that are used for interaction with the Timer Cache of section 3.3. These are the *Checkout Timer Cache* and *Update Time Cache* subroutines of sections 3.5.1 and 3.5.4 respectively.

[40] If the Core ID is invalid and cannot be turned into a core_index then the OPC_ERR bit of the STATUS register is set.
[41] Any number of LUC engines can process different operations, even if the workspace size is 2KB.

- Sections 3.5.5 and 3.5.6 describe the subroutines *Send Workspace* and *Modify Workspace Validity*. These are used to send a workspace to a Processor Core, and then to modify the validity of that workspace.
- Section 3.5.8 on the *LUC Engine Start* is the main flow chart that the LUC Engines use. This is the starting point for LUC Engine processing.
- Sections 3.5.9 through 3.5.20 describe the processing for the various LUC commands. A software orientated overview of these commands is given in section 2.4.

3.5.1 Checkout Timer Cache Subroutine

The flow chart of Figure 17 defines the behaviour for updating the timer cache of the LUC. The operation is relatively simple: we first issue a lock unconditional command to the timer cache unit using the socket ID as the key. The timer will respond to this command when a locked timer cache line is available, creating a line if required. If a line was created then we populate it using the timer table in DDR memory. We then update the checkout bitmap to indicate that this socket ID is now checked out, and finally issue an unlock command to the timer cache, causing the lock bit to be cleared. For more details on the timer cache commands please see section 3.3.

Notice that throughout the modification of the timers in the timer cache we need not worry about other units accessing the same timers: the lock bit in the timer cache takes care of this for us.

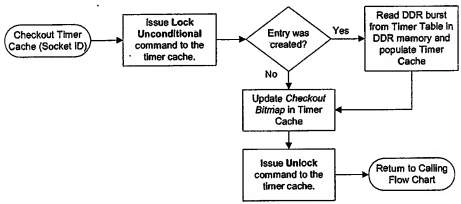

Figure 17: Checkout Timer Cache Subroutine

3.5.2 Obtain Timer Lock Subroutine

The *Obtain Timer Lock* subroutine of Figure 18 is used to obtain a timer lock on an entry that has been previously checked out. First we use issue the lock unconditional command to the timer cache. If the response from the timer cache indicates that an entry was just created then we must have an error condition[42]: a timer cache entry should have been created when the flow was checked out.

[42] We must have an error condition since the *Obtain Timer Lock* subroutine is **only** used when we know the flow has been previously checked out, in which case it **must** have a Timer Cache line. We recover from this error condition by simply setting an error bit and continuing as normal.

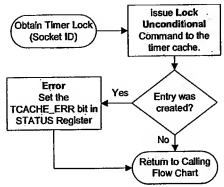

Figure 18: Obtain Timer Lock Subroutine

3.5.3 Retire Timer Lock Subroutine

The *Retire Timer Lock* subroutine of Figure 19 is used to retire a lock for a flow in the Timer Cache. The first step we take is to use the Socket ID to determine which bit in the checkout bitmap corresponds to this flow: that bit is then cleared. If the resulting checkout bitmap is zero then no other flows are checked out on this timer cache entry and it should be written back to the DDR timer table. An unlock and remove command is then issued to the timer cache. If the resulting checkout bitmap is not zero then we simply issue an unlock command to the timer cache.

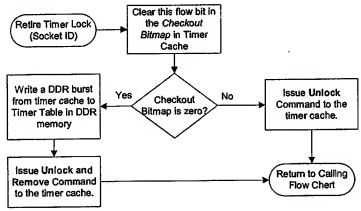

Figure 19: Retire Timer Lock Subroutine

3.5.4 Update Timer Cache Subroutine

Figure 20 illustrates the steps that must be performed when updating the timer cache with values from a workspace that has just been checked in. Note that this flow chart assumes that the appropriate locks have already been applied for in the Timer Cache.

First we update the timers with the timers in the workspaces. We do this using the *Write Timer Bitmap* of the workspaces. If bit i of this bitmap is set then we must update timer i in the timer cache. See Table 34 for a mapping from timer ID to timer. Note that no timer expirations can be issued while the timers are updated: we must wait until the flow is fully checked in before we can do this, i.e. we must have deleted the entry from the FDC. Note that two workspaces can be written back to the LUC, so we must apply some logic to merge the timers.

To merge the timers, using the READY bitmap described in section 3.4.2, we see if the Protocol Core has written back a workspace. The READY bitmap is indexed using the Protocol Core Index and the Protocol Core Workspace ID, both supplied as parameters to the *Update Timer Cache* subroutine. The Protocol Core ID is turned into a Protocol Core Index using Table 2. If the READY bitmap indicates that this workspace has been written back, then we mask the *Write Timer Bitmap* of the workspace with the PC_TMR_MASK value from the Extended Configuration (EXT_CONFIG) register[43]. This register allows the software to restrict which timers a Protocol Core can modify. This masked bitmap is then used to determine which timers should be updated due to the Protocol Core.

If workspace splitting is enabled, and the Interface Core has written back a workspace, then the same operation as above are applied to the Interface Core timers. Note that this means that Interface Core timers will overwrite Protocol Core timers if the *Write Timer Bitmaps* overlap, and the IC_TMR_MASK allows it.

Also note that if neither the Protocol Core nor the Interface Core has written back a workspace, then both READY bits will be zero. In that case no timers will be adjusted.

Notice that throughout the modification of the timers in the timer cache we need not worry about other units accessing the same timers: the lock bit in the timer cache takes care of this for us.

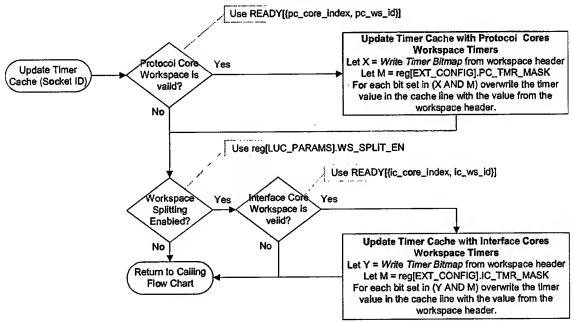

Figure 20: Update Timer Cache Subroutine

3.5.5 Send Workspace Subroutine

This subroutine is used to send a workspace to a Protocol Core or a {Protocol Core, Interface Core} pair. It assumes that a workspace header has already been partially built by the calling flow chart. This description of this subroutine assumes that the reader is familiar with flow state splitting as described in sections 2.3.5.3 and 2.3.5.4.

[43] To determine where the *Write Timer Bitmap* is in the LUC Workspace In Buffer use the same logic as described in Figure 16.

This subroutine takes a parameter that indicates if it is to build a header only message, a flow state workspace or a listen state workspace. Based on what type of message we are sending, we initialise the variables described in Table 22.

Field	Description
sh_size	The sh_size variable is the size of the *Shared Area* of Figure 10 in terms of 128-bit words. Note that this includes 32-bits of data that is used for the *Flow State Write Bitmap*. In the case where only the header is being sent, the value of this variable is forced to zero. If a workspace header and payload are being sent, then this variable is initialised to the appropriate field of the Extended Configuration register (see section 4.8.58).
pca_len	This is the number of 128-bit words that hold the *Processor Core Area* as defined in Figure 9 and Figure 10. Note that this is different from the pc_len variable below, since pc_len includes the workspace header and any *Shared Area*. This variable is initialised using fields from the Extended Configuration and LUC Parameters registers (see sections 4.8.58 and 4.8.9).
pc_len	This is the number of 128-bit words that will be sent to the Protocol Core. It includes the 2 x 128-bit words for the workspace header, any *Shared Area* and the *Protocol Core Area*. In the case where only the header is being sent, the value of this variable is forced to zero. If a workspace header and payload are being sent, then this variable is initialised using fields from the Extended Configuration and LUC Parameters registers (see sections 4.8.58 and 4.8.9).
ic_len	This is the number of 128-bit words that will be sent to the Interface Core. It includes the 2 x 128-bit words for the workspace header, any *Shared Area* and the *Interface Core Area*. In the case where only the header is being sent, the value of this variable is forced to zero. If a workspace header and payload are being sent, then this variable is initialised using fields from the Extended Configuration and LUC Parameters registers (see sections 4.8.58 and 4.8.9).
ddr_socket_base	This is a DDR pointer that points to the location where the flow state is being held. This is calculated using the Socket ID and the base of the flow state table (see register 4.8.18). This is a pointer to a 64-byte word.

Table 22: Send Workspace Subroutine Variables

Next we determine if workspace splitting is enabled. If it is not enabled then we first mark that workspace as no longer being ready. This allows us to determine at a later date whether this workspace has been written back. It is important to do this *before* we send out the workspace itself[44]. We then adjust the workspace header with the Protocol Core ID and Protocol Workspace ID, and send pc_len 128-bit words (which is pc_len / 4 64-byte words) to the designated Protocol Core via a single Message Bus transaction. Note that in this case the Workspace Header actually overwrites part of the data that is read from DDR memory. The reason for this is that the Protocol Core portion of the flow state memory actually contains the Workspace Header, as illustrated in Figure 9 and Figure 10.

If workspace splitting is enabled then we first mark both workspaces as no longer being ready. Again, it is important to do this before the workspaces are sent. We then adjust the workspace header with the Protocol Core ID and Protocol Workspace ID, and send pc_len 128-bit words (which is pc_len / 4 64-byte words) to the Protocol Core. Again, the Workspace Header actually overwrites part of the data that is read from DDR memory.

We must now determine if there is any *Shared Area* in the flow state. If the *Shared Area* size is zero then we simply send ic_len 128-bit words (which is ic_len / 4 64-byte words) from (ddr_socket_base + pc_len / 4) to the Interface Core with the Interface Core ID and Interface Workspace ID written into the workspace header. The value (ddr_socket_base + pc_len / 4) comes from the fact that the *Interface Core Area* directly follows the *Protocol Core Area*. If a *Shared Area* size is non-zero then we must construct the Interface Core workspace from two regions. The first region is (2 + sh_size) 128-bit words from the start of the socket (ddr_socket_base). The second region is the *Interface Core Area*, which is read from the same location as when sh_size = 0. Once those two regions have been concatenated, they are sent as a single Message Bus transaction with the Interface Core ID and Interface Workspace ID written into the workspace header.

[44] The READY bitmap **must** be cleared before the workspace is sent out. It should not be cleared after each USID, ULID, TDFK or TDLK is issued. See section 3.4.2 for further information.

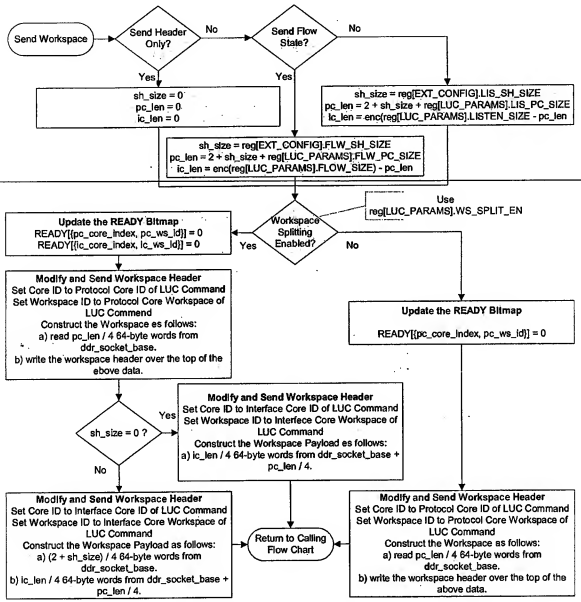

Figure 21: Send Workspace Subroutine

3.5.6 Modify Workspaces Validity Subroutine

The *Modify Workspaces Validity* subroutine of Figure 22 is used by the LUC to modify the validity of the Protocol Core and Interface Core workspaces. This is used when processing the update and teardown commands of the LUC. The workspace is marked as invalid just before the LUC issues the RMFIDX command to the FDC. If the FDC response indicates that events are still pending for that flow then the LUC needs to reset the state of the workspace to valid without writing the whole workspace.

The LUC sets and clears the validity of a workspace using message bus transactions with the appropriate ws_v value. The ws_v value of 2'b11 indicates that the workspace should be marked as valid, and 2'b10 indicates that a workspace should be marked as invalid. The first step of the *Modify Workspace Validity* flow chart is to set the *val* variable to the correct value depending on whether the callee wants to set or clear the workspace validity.

First we modify the validity of the Protocol Core. If workspace splitting is enabled we then modify the validity of the Interface Core. Note that in both cases we wait for the message bus transactions to complete – we are not allowed to return from this subroutine until we know that the set / clear of the workspace validity has made it all of the way to the Protocol Cluster.

If workspace splitting is not enabled then we only modify the workspace validity on the Protocol Core.

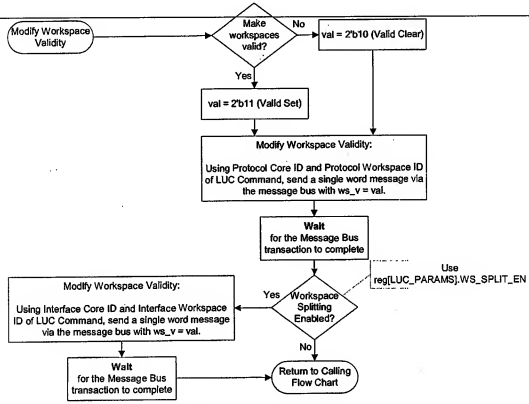

Figure 22: Modify Workspaces Validity Subroutine

3.5.7 Write Workspace Subroutine

The *Write Workspace* subroutine of Figure 23 is used to merge the data from up to two workspaces into a format that can be written back to the DDR memory. Table 23 defines the variables that are used in this subroutine. The sh_size, pca_len, pc_len and ic_len variables that are used are the same values defined in Table 22.

This section assumes that the reader is familiar with the workspace and flow state layouts described in sections 2.3.5.3 and 2.3.5.4.

Field	Description
ddr_socket_base	This is a DDR pointer that points to the location where the flow state is being held. This is calculated using the Socket ID and the base of the flow state table (see register 4.8.18). This is a pointer to a 64-byte word.
pc_core_index	The Protocol Core Index. This is an encoded value of the Protocol Core ID. See section 2.1. If the Protocol Core ID is invalid and cannot be turned into a Core Index then the OPC_ERR bit of the STATUS register is set.
ic_core_index	The Interface Core Index. This is an encoded value of the Interface Core ID. See section 2.1. If the Protocol Core ID is invalid and cannot be turned into a Core Index then the OPC_ERR bit of the STATUS register is set.
lwib_pc_base	This is the same as the core_base of Table 21.
lwib_ic_base	This is the same as lwib_pc_base, except it is for the Interface Core Index.
lwib_pc_shift	This is the same as the core_shift of Table 21.
lwib_ic_shift	This is the same as the lwib_pc_shift field, except it is for the Interface Core Index.
lwib_pc_addr	This is an address into the LUC Workspace In Buffer that points to the start of the workspace. It is calculated from lwib_pc_base and lwib_pc_shift. This is a pointer to a 128-bit word.
lwib_ic_addr	This is the same as the lwib_pc_addr field, except it is for the Interface Core Index.
pc_mask	This is the mask that should be applied to the Protocol Cores *Flow State Write Bitmap* before it is used. It implements the memory protection described in section 2.3.5.7. Depending on whether a flow is listen state is being written, this value comes from the appropriate PC_FLW_WR_MASK or LIS_WR_MASK registers (see sections 4.8.59 and 4.8.61).
ic_mask	This is the same as the pc_mask except that it is for the Interface Core.

Table 23: Write Workspace Subroutine Variables

The first step of the *Write Workspace Subroutine* is to calculate the variables defined above. Some of these variables depend on whether a listen state or flow state is being written – the caller passed this information into the subroutine. We then use the READY bitmap to determine if the Protocol Core workspace has been written back. If it has been written back then the appropriate READY bit will have value 1. See section 3.4.2 for more information on the READY bitmap.

If the Protocol Core workspace has been written back to the LUC then we fetch the *Flow State Write Bitmap* from the appropriate location in the LUC Workspace In Buffer. This field is in the 128-bit word pointed to by (lwib_pc_addr + 2). We then write back each 64-byte block that is set in (fswb AND pc_mask) to the correct place in the flow state. As illustrated in Figure 9 and Figure 10, the Protocol Core's workspaces maps onto the front of the flow state, so each 64-byte block at address (lwib_pc_addr + i * 4) in the LUC Workspace In Buffer is written to address (ddr_socket_base + i) in the DDR memory. Note that this works regardless of whether workspace splitting is enabled or not.

We must then determine if we need to merge in the workspace from an Interface Core. If workspace splitting is enabled, and the Interface Core workspace has been written back then data needs to be copied from the Interface Core workspace into the flow state. We do this by looking at the *Flow State Write Bitmap* for the Interface Core, which is located in the 128-bit word pointed to by (lwib_ic_addr + 2). For each bit, i, in this field, there are two cases:

1. The value of i is less than (sh_size + 2) / 4. In this case we must be writing the *Shared Area*. The *Shared Area* is located at the front of the flow state, so we write each 64-byte block from (lwib_ic_addr + i * 4) in the LUC Workspace In Buffer to (ddr_socket_base + i) in the DDR memory. We add 2 to the LUC Workspace In Buffer address since the Workspace Header of the Interface Core is not included in the flow state.

2. The value of i is greater than or equal to (sh_size + 2) / 4. In this case we must be writing the *Interface Core Area*. As illustrated in Figure 9 and Figure 10, the *Interface Core Area* of the Interface Core's workspaces maps onto the end of the flow state, so each 64-byte block at address (lwib_ic_addr + i * 4 – pca_len) in the LUC Workspace In Buffer is written to address (ddr_socket_base + i) in the DDR memory. Note how we subtract pca_len from the address in the

LUC Workspace In Buffer. This is due to the alignment of the *Flow State Write Bitmap* for an Interface Core.

Note that it is the Interface Core's shared area that is written last. This means that if both the Protocol Core and the Interface Core write to the same *Shared Area*, then the Interface Cores value will be the one written.

As specified in section 2.3.5.6, if a workspace is being updated then either the Protocol Core or the Interface Core must have written back at least the workspace header. In fact, if this is the first update of the flow then it must be the Protocol Core that does the update, and at least the first 64-bytes must be written back (with the appropriate bit set in the writeback bitmap). If this rule is not adhered to then the LUC will set the GEN_ERR bit of the STATUS register while processing the flow chart below.

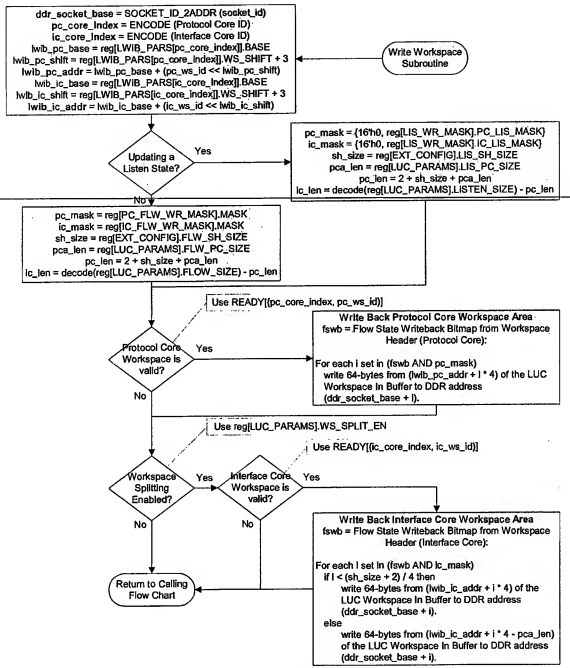

Figure 23: Write Workspace Subroutine

3.5.8 LUC Engine Start

The *LUC Engine Start* flow chart of Figure 24 is the main entry point for each LUC Engine. The task is relatively obvious: we wait for a command from the Dispatcher, and depending on the command we update

the appropriate counter and then continue with a flow chart appropriate to that command. If we do not recognise the command from the Dispatcher then an error bit (OPC_ERR) is set in the LUC status register.

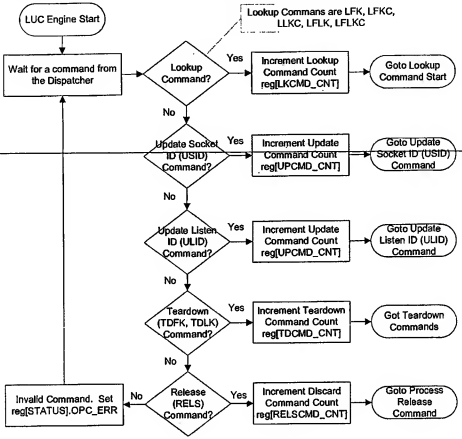

Figure 24: LUC Engine Start

3.5.9 Lookup Commands (LFK, LFKC, LLKC, LFLK, LFLKC)

Figure 25 Illustrates the steps that must be followed for the lookup commands. First we determine if the command is lookup with listen key create (LLKC), setting the listen key appropriately. In either case we then compute the hash value for the flow key and the listen key. Next we apply and wait for locks from the flow locker. To determine which locks should be applied for see Table 4.

If the command is LLKC then we are searching for a listen key and we continue with the *Process Listen Lookup* flow chart of Figure 27. Note how for an LLKC command we search for the *exact* value of the flow key, but when calculating the hash we use the flow key masked with the listen key. This allows the listen fixers described in section 2.5.3 to be inserted into the LUC. If we are searching for a flow key then we chain down the hash list indicated by the hash value (FH) looking for a match.

If we are searching for a flow key, and a match was found, then we must check the flow out. To do this we first of all use the *Checkout Timer Cache* subroutine of section 3.5.1 to ensure that the timer table line for this flow is present in the timer cache. We then create a workspace and send it to the appropriate processor cores using the *Send Workspace* subroutine of section 3.5.5. After the workspace has been sent we release the locks from the flow locker.

If a flow key was not found in the hash table then we continue processing with the *Process Flow Not Found* flow chart of section 3.5.10.

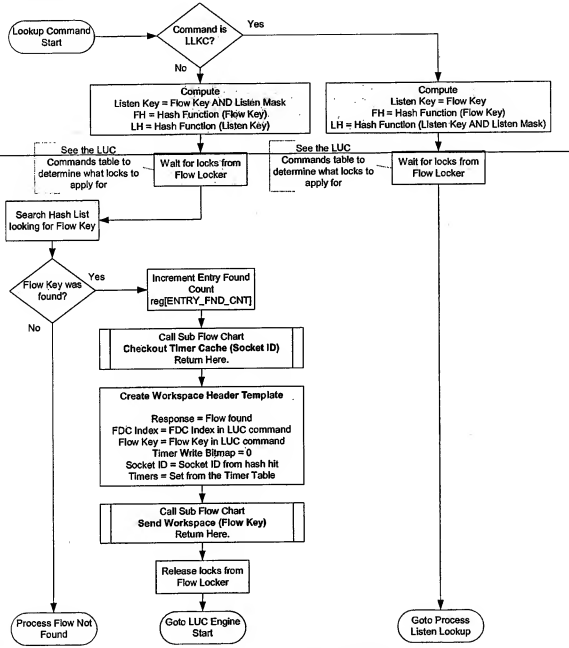

Figure 25: Lookup Command Flow Chart

3.5.10 Process Flow Not Found

Figure 26 illustrates the steps that must be performed when a lookup command did not find a match on the flow key. The commands that this flow chart must process are LFK, LFKC, LFLKC and LFLK. If the

command is an LFK command then we simply report to the processor cores that a flow key was not found, and then release the locks from the flow locker.

If the command is not an LFK command then we check to see if it is a LFKC and that TACL is enabled for LFKC commands. If this is the case then we must perform a TACL lookup before creating a flow entry. If the TACL lookup allows us to create an entry then we do so using the *Process Flow Create* flow chart of Figure 29. If the TACL lookup denied us then we report this information to the Processor Cores. For more information on TACL lookups see section 2.5.

We must now check if the command is an LFLK or LFLKC command. If it is not then the command must be an LFKC command[45], so we continue with the *Process Flow Create* flow chart of Figure 29. If the command was an LFLK or LFLKC then we continue with the *Process Listen Lookup* flow chart of Figure 27.

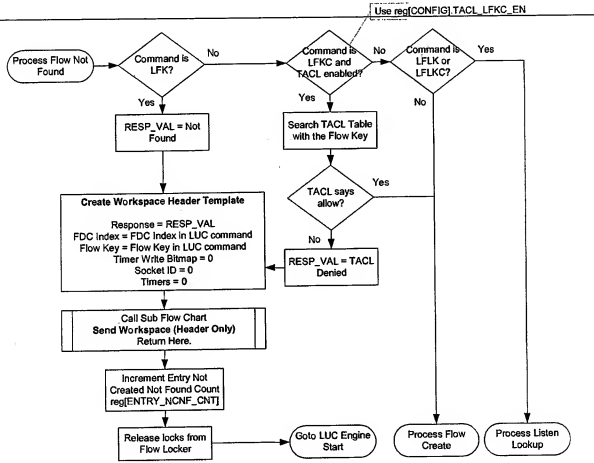

Figure 26: Process Flow Not Found

3.5.11 Process Listen Lookup

The *Process Listen Lookup* flow chart of Figure 27 is used to perform a lookup of the listen key for the LFLK, LFLKC and LLKC commands. We first step we take is to check if the command is LFLKC and if TACL is enabled for that command. If it is enabled then we perform a TACL lookup using the flow key. If the TACL

[45] It must have been an LFKC command *and* TACL was disabled.

lookup returns a deny response, then this is reported to the Processor Cores and we continue back to the *LUC Engine Start* flow chart. If the TACL lookup response is an allow, or if this is not an LFLKC command with TACL enabled, then we perform a lookup of the listen key in the hash table. Note that if a TACL lookup was performed then the Listen Key may be modified using the *Listen Fixer* and *Listen Fixer Pointer* fields of the TACL entry.

If the listen key is found then we continue with the *Process Listen Key Found* flow chart of Figure 28. If the listen key is not found then we must check whether the command was a LLKC. If it was then we continue with the *Process Listen Create* flow chart of Figure 30. If it was not an LLKC then it must have been a LFLK or LFLKC command. In either case we respond with a Not Found response to the Processor Cores, and then continue with the *LUC Engine Start* flow chart.

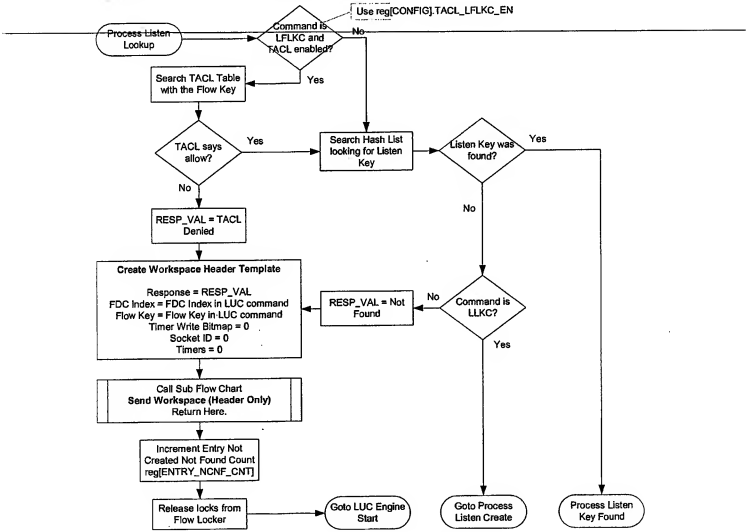

Figure 27: Process Listen Lookup

3.5.12 Process Listen Key Found

Figure 28 illustrates the steps that are taken when a listen key has been found. This flow chart can be reached when the LLKC, LFLK or LFLKC commands are executed. If this command is the LFLKC command then since we have found a listen entry we are allowed to create a flow entry, so we continue with the *Process Flow Create* flow chart of section 3.5.13. If this is the LFLK or LLKC command then we simply report to the protocol core that a listen entry was found, and release the locks from the flow locker.

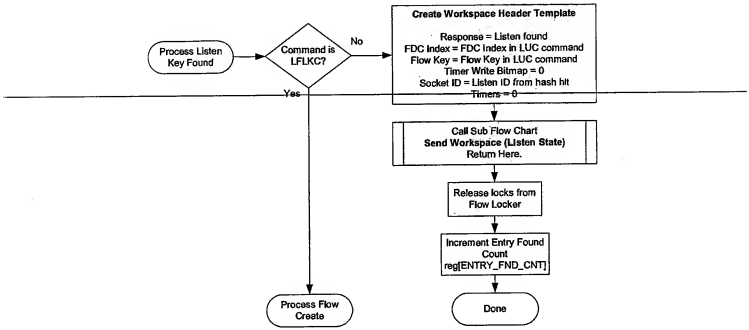

Figure 28: Process Listen Key Found

3.5.13 Process Flow Create

The steps of Figure 29 are used when a flow needs to be created. The first thing we must do is to check whether adding this flow entry to the hash list will cause the hash list to grow larger than the maximum allowed value. If this is the case then we report that fact to the Processor Cores, and continue with the LUC Engine Start flow chart.

Assuming that the hash list has not reached its maximum length, we attempt to create the flow in the hash table with the Entry Is Flow (EIF) set to one. We then determine if the entry really was created: if it was not created then we create a workspace indicating this and send it to the Processor Cores.

If the entry was created then we must checkout the timers into the timer cache. This is done using the *Checkout Timer Cache* subroutine of Figure 17.

The next check determines if we are processing the LFKC command. If we are processing this command then we create a workspace indicating that a new flow was created, and write the workspace (header only) to the associated Processor Cores.

If the command is not LFKC then it must be LFLKC, in which case we create a workspace that indicates that a new flow was created due to a listen match. Again, this workspace is written to the associated Processor Cores, but this time we send the listen state as well. Note that in both cases we set the timers to zero, i.e. disabled.

In all cases we release the locks from the flow locker and continue with the *LUC Engine Start* flow chart.

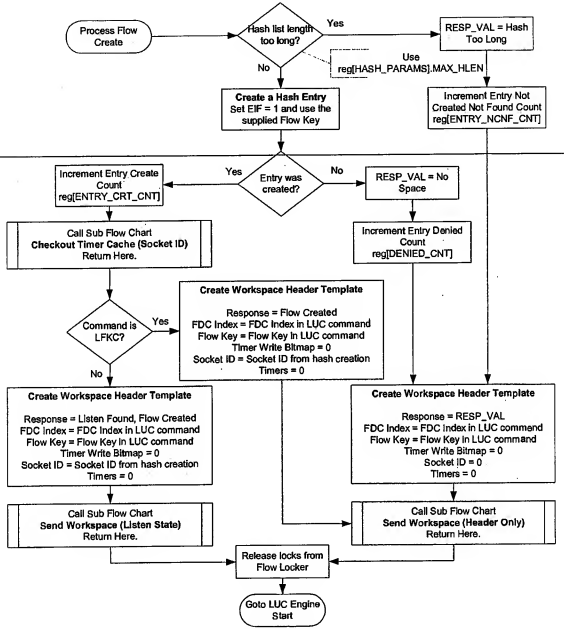

Figure 29: Process Flow Create

3.5.14 Process Listen Create

Figure 30 illustrates the steps that are taken when creating a listen entry. We must first check that adding this listen entry will not cause the hash list to go beyond its maximum length. If the hash list would grow too large then we report this fact to the Processor Cores and then continue with the *LUC Engine Start* flow chart.

Next we determine if the hash list would grow too large if the listen entry were added. If this would be the case, then we send a *Hash Too Long* response to the Processor Cores. Assuming the hash list would not grow too large, we must now checked if TACL is enabled. If it is then we do a TACL lookup and respond with *TACL Denied* if we do not get an allowed response.

Assuming the TACL response allows or, or if TACL is not enabled for LLKC, we then create a hash entry with the listen key and with the Entry Is Flow (EIF) flag set to zero. Note how we create a listen entry but we use the flow key rather than the listen key (which is a masked flow key). The reason for this is that we must be able to create listen entries that have bits set outside of the Listen Mask area. We need to do this since the Listen Fixer may adjust these fields when used for an LFLKC or LFLK command. For the LLKC command, which we must be processing, listen fixers are not used[46]. Depending on the success of this hash entry creation, we create and send a workspace to the associated Processor Cores. The response in the workspace indicates if the listen entry was created or not. We then release all locks from the flow locker and continue with the *LUC Engine Start* flow chart.

[46] It is assumed that the unused fields of the flow key have been zeroed out to match the fields that are not fixed by a listen fixer and are not included in the Listen Mask.

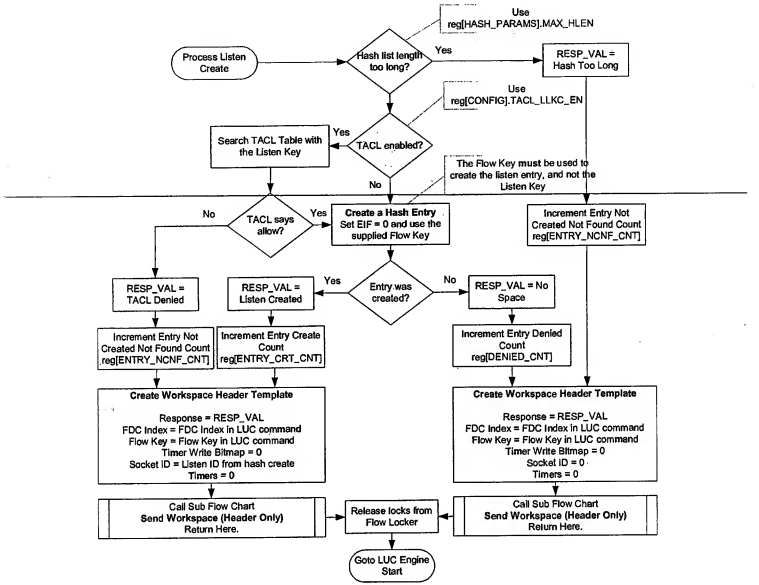

Figure 30: Process Listen Create

3.5.15 Lookup with Socket ID (LSID)

The flow chart of Figure 31 is used when the LUC receives a lookup with socket ID (LSID) command. This is a very simple command to execute, and it requires no access to the hash table. We simply use the socket ID to determine where the flow state is being held for this flow. We then create a workspace and transmit it to the requesting Processor Cores.

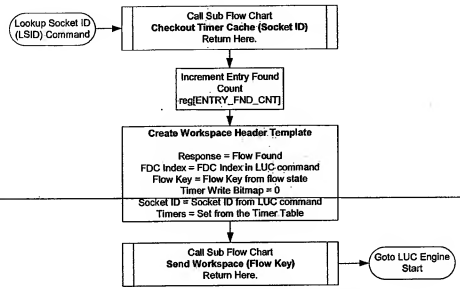

Figure 31: Lookup Socket ID (LSID) Flow Chart

3.5.16 Update with Socket ID (USID)

Figure 32 illustrates the steps that are taken when the LUC receives an update with Socket ID command. First we write the data in the workspace in buffer back to the DDR flow state memory using the *Write Workspace* subroutine of section 3.5.7. The next two tasks can them be performed in parallel. First we must clear the valid bit in the workspace of the Processor Cores using the *Modify Workspace Validity* subroutine of section 3.5.6. Note that this subroutine does not return until the validity has been modified, which is important since we cannot issue the RMFIDX command until after that. In parallel with this operation, we can update the timer cache using the *Update Timer Cache* subroutine. However, before we do that we must obtain a lock in the Timer Cache using the *Obtain Timer Lock* subroutine of section 3.5.2. Note that after processing the *Obtain Timer Lock* subroutine we still have a lock on the Timer Cache, therefore preventing any CRTIMER commands from being issued by the TCU.

When both of these operations have completed, we can issue a command to the FDC. The important thing is that the workspace valid bit must be set to zero on the Processor Cores before the RMFIDX command can be issued to the FDC. If this is not done then there is a race condition where a new flow could be assigned to that workspace, but the workspace valid bit has not been reset from the previous flow.

After the RMFIDX command has been issued to the FDC we must examine the FDC response. If the FDC has indicated an error then we set the appropriate bit in the STATUS register. If this response indicates that the FDC entry is not in the RECEIVED state then there are no more outstanding events for this flow. In this case we use the *Retire Timer Lock* subroutine of section 3.5.3 to update the *Checkout Bitmap*, unlock the Timer Cache line, and possibly write it back to DDR memory.

If the FDC response indicates that the FDC entry is in the RECEIVED state then new events must have arrived after the Dispatcher issue the USID command. We must therefore set the valid bit in the workspace to 1, and unlock the Timer Cache line. Note that in simply unlocking the Timer Cache line we do not update the *Checkout Bitmap*, i.e. it is left in the checked out state. The scenario where the FDC entry will be found in the RECEIVED state is when the Dispatcher receives an event just after having sent the LUC USID

command. In this case the Dispatcher will mark the FDC entry as pending, and forward the event to the protocol core[47].

Note that through out the processing of the USID command there was a lock on the Timer Cache. The Timer Control Unit (TCU) can therefore not issue a CRTIMER command during this period.

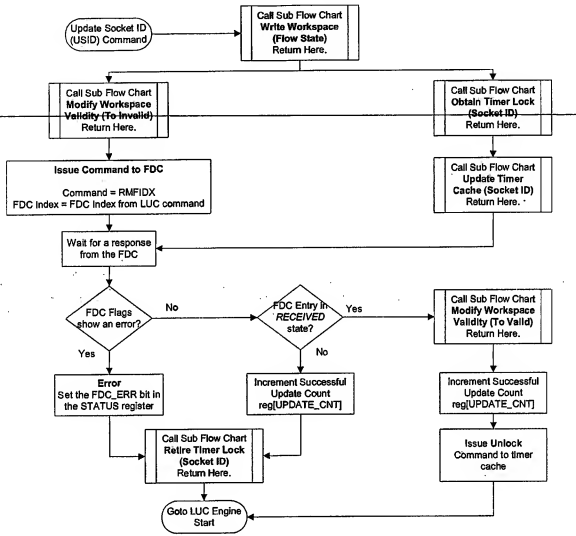

Figure 32: USID Flow Chart

3.5.17 Update with Listen ID (ULID)

Figure 33 performs the same type of function as the USID command, except this time it is a listen ID that is being updated. Since listen sockets do not have any associated timers, we do not have to worry about updating the timer cache.

[47] The Dispatcher guarantees that only one USID command will be outstanding per flow. It does this by delaying the update of the FDC if the protocol core replies with a done event before the previous update has finished.

The first step that we take is to write the listen state back to the DDR memory using the *Write Workspace* subroutine of section 3.5.7 and then reset the valid bit of the workspace using the *Modify Workspace Validity* subroutine of section 3.5.6. Note that the *Modify Workspace Validity* subroutine will not return until the validity of the state has been modified, i.e. until that transaction has made it all of the way across the Message Bus.

We then issue an RMFIDX command to the FDC and wait for the response. If the FDC has indicated an error then we set the appropriate bit in the STATUS register. If the response indicates that the FDC entry is not in the RECEIVED state then the ULID command has completed. If the response indicates that the FDC entry is now in the RECEIVED state then we must set the valid bit of the workspace back to 1. The scenario of when the FDC entry will be found in the RECEIVED state is the same as the scenario described for the USID command.

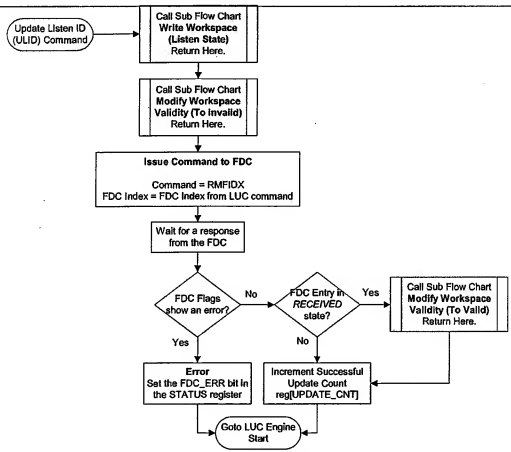

Figure 33: ULID Flow Chart

3.5.18 Teardown Commands (TDFK, TDLK)

Figure 34 illustrates the steps that are taken when a teardown with flow key (TDFK) or teardown with listen key (TDLK) command is received by the LUC. The important thing we should note about both of these commands is that unlike the update commands (USID, ULID), it is not possible for the FDC entry to be in anything other than the CHECKED IN state after the RMFIDX command has been issued. The reason is that when the Dispatcher issues a teardown command to the LUC the FDC entry is in the DELETE state.

When it gets into this state the only possible transition is to CHECKED IN. See the Dispatcher and FDC HLDs for further details.

The first step in the teardown command flow chart is to determine the listen key value, flow key hash and listen key hash. For teardown commands we use the full flow key as the listen key without any masking. This allows us to tear down listen entries that have been fixed (see section 2.5.3). However, for the listen hash we use the flow key masked with the listen key. Again, this is required to support listen fixers, where the hash is always applied without the listen fix in place. Next we apply for locks on the flow locker. See Table 4 for a description of the locks required for the TDFK and TDLK commands. We then search for the flow/listen key in the hash table. If a flow key is found then it is deleted, otherwise we set an error bit in the STATUS register.

We then examine the LUC command to see if it is a teardown with listen key (TDLK). If it is then there is no need to perform any timer management, so we skip directly to the *Process Delete Entry* flow chart.

Assuming the command is not a TDLK, we use the *Obtain Time Lock* subroutine to obtain a lock on the Timer Cache line. Next we disable all of the timers for this flow in the timer cache. We then use the *Retire Timer Lock* subroutine of section 3.5.3 to update the *Checkout Bitmap*, unlock the Timer Cache line, and possibly write it back to DDR memory.

In all cases we then continue with the *Process Delete Entry* flow chart. Note that in both the flow chart below, and the *Process Delete Entry* flow chart, no access is made to the LUC Workspace In Buffer, i.e. it is not required that the Processor Cores write back a workspace for a tear down command.

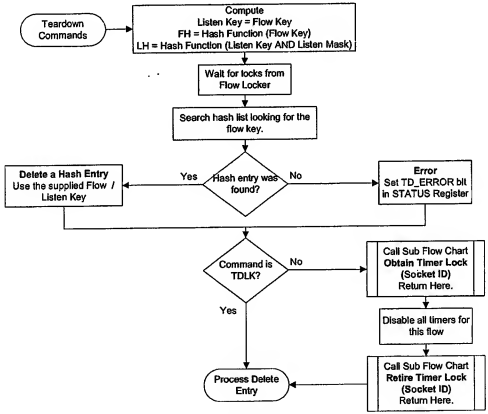

Figure 34: Teardown Commands (TDFK, TDLK) Flow Chart

3.5.19 Process Delete Entry

Figure 35 is a continuation of the flow chart of Figure 34. It is only used when a teardown command has been issued to the LUC. It assumes that the timer cache has already been updated, and any appropriate locks are in place.

The first task we perform is to set the valid bit of the workspace in the Processor Cores to zero using the *Modify Workspace Validity* subroutine of section 3.5.6. Note that the *Modify Workspace Validity* subroutine will not return until the validity of the state has been modified, i.e. until that transaction has made it all of the way across the Message Bus. We then issue the RMFIDX command to the FDC and wait for its response.

When the response from the FDC is available we check that the FDC entry is in the CHECKED IN state. As noted at the start of this section, if the entry is not in the CHECKED IN state then an error has occurred, and the appropriate error bit is set[48].

Finally we release all locks that were obtained. The teardown command has then been executed.

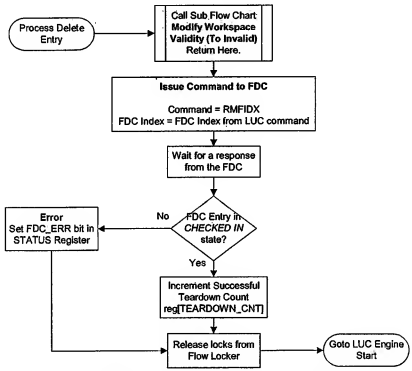

Figure 35: Process Delete Entry Flow Chart

3.5.20 Process Release (RELS) Command

Figure 36 defines the steps that are required for the LUC to process a RELS command. This command is used by the Processor Cores to release a workspace that can neither be updated or torn down, e.g. a workspace that had a *Not Found* response. See section 2.4.12 for more details on the RELS command.

Processing for the RELS command is very simple. We start by calling the subroutine that marks a workspace as invalid. This prepares the workspace for the next flow. We then issue the RMFIDX command

[48] This includes the case when the FDC entry is in the RECEIVED state, or the FDC responds with an error.

to the FDC. Since the FDC entry should be in the *DELETE* state[49], the FDC entry should be in the *CHECKED IN* state after the RMFIDX command. We check that this is the case and set the appropriate error bit if it is not[50]. We then increment a counter and continue with the *LUC Engine Start* flow chart.

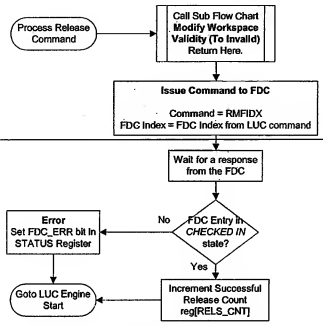

Figure 36: Process Release (RELS) Command

3.6 Timer Control Unit (TCU)

3.6.1 TCU Variables

The Timer Control Unit (TCU) of the LUC controls the expiration and ticking of the Timer Table. In the following sections we examine the logic that is required to perform this operation.

The goal of the TCU is to tick each timer in the Timer Table of section 2.6.2. Some timers in that table have a resolution of 200ms, while the others have a resolution of 2s. The TCU performs its function by examining sequential Timer Table entries every 200ms / N_{FLOWS} seconds. We therefore know that after N_{FLOWS} ticks of the TCU clock, 200ms has passed. The period of the TCU clock is programmed into the INTERVAL value of the TCU_CTRL register (see section 4.8.10).

This obviously accommodates the 200ms timers, but what about the 2s timers? The TCU accomplishes this task by hitting the 2s timers on every 10th access to the Timer Table. Essentially there is a MOD 10 counter (M10CNT) that is incrementing for each TCU tick, and depending on the value of that counter different Timer Table entries have their 2s timers examined. Table 24 defines which Timer Table indexes have their 2s timers ticked dependent upon the value of M10CNT.

We now know, based on the M10CNT value, which elements in the timer table should have a 2s tick applied. The question is how to we iterate through those indexes? To do this we use another MOD 8 counter

[49] The Dispatcher HLD states that a RELS command should be setup like a teardown, i.e. it should use TDFIDX.
[50] This includes the case when the FDC entry is in the RECEIVED state, or the FDC responds with an error.

(M8CNT) that is initialised to a different value each time the M10CNT is changed. For each element we visit in the timer table we increment M8CNT. Each time M8CNT equals zero we apply a 2s tick.

Table 24 illustrates these two counters that are used by the TCU. For example, on the very first pass of the timer table M10CNT will have value 0 and M8CNT will be initialised to 0. Therefore, Timer Table entries 0, 8, 16, ... will have a 2s tick applied. After that first pass of the table M10CNT will be set to value 1, and M8CNT will be initialised to value 1. The 2s tick will therefore be applied to elements 7, 15, 23, ... etc.

MOD 10 Counter (M10CNT) Value	Timer Tables entries that have a 2s tick	MOD 8 Counter (M8CNT) Initialisation Value when MOD10 Counter is changed.
0	0, 8, 16, ...	0
1	7, 15, 23, ...	1
2	6, 14, 22, ...	2
3	5, 13, 21, ...	3
4	None	N/A
5	4, 12, 20, ...	4
6	3, 11, 19, ...	5
7	2, 10, 18, ...	6
8	1, 9, 17, ...	7
9	None	N/A

Table 24: TCU M10CNT and M8CNT Counters

Another variable that the TCU uses is the *CurrentTmrIdx*. This is an index into the Timer Table that is incremented for each TCU tick.

3.6.2 Timer Overflow

In the following sections we will define what actions the TCU takes on each of its internal ticks. These operations will require locking the Timer Cache, interaction with the FDC and other tasks. If, for some reason, these tasks take longer than the TCU tick period (reg[TCU_CTRL].INTERVAL) then the TCU uses an 8-bit counter to indicate that it missed a tick. While this count is non-zero, the TCU will always issue a tick, decrementing the counter value. If this counter overflows then the TCU has effectively slipped time. If this occurs then the TCU sets the TIME_OVFL bit in the status register. Note that the TCU will continue to function correctly, the only down side is that a tick was missed.

3.6.3 Start TCU Flow Chart

The *Start TCU Flow Chart* of Figure 37 is the main starting point of the TCU. The first thing this routine does is to wait for a number of clock ticks, as indicated by the INTERVAL value of the TCU_CTRL register (see section 4.8.10). Once that period has expired, it uses the CurrentTmrIdx value to issue a Lock Unconditional request to the Timer Cache. When this lock is granted to the TCU, it has complete access to a timer cache line. If a cache line was created then the TCU must fill it from the Timer Table in DDR memory.

Note that we now have either four or eight flows worth of timers in the single Timer Cache line, depending on whether this is a B10 or S10 LUC. We now decrement the value of each 200ms timer in the Timer Cache line using the *Decrement Timers* subroutine of section 3.6.5. The *Decrement Timers* subroutine does not trigger any events, it simply decrements the timer values.

We must now determine if we need to decrement the 2s timers that are in the Timer Cache line. As described in section 3.6.1, this is done using a modulo ten counter, M10CNT, and a modulo eight counter, M8CNT. If the value of the M8CNT variable is zero, and if the value of M10CNT is not 4 or 9, then we should also tick the 2s timers. Again, this is done using the *Decrement Timers* subroutine.

Finally, we increment the value of the modulo eight counter M8CNT, and then continue with the *Process Ticked Timers* flow chart of Figure 38.

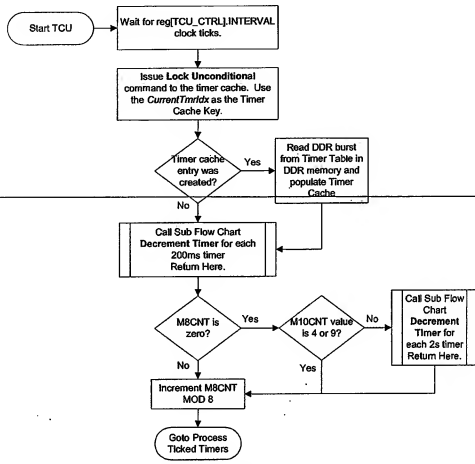

Figure 37: Start TCU Flow Chart

3.6.4 Process Ticked Timers

The *Process Ticked Timers* flow chart of Figure 38 is used after a Timer Cache line has been filled with an entry from the Timer Table, and the timer values within that cache line have been decremented. Note that this cache line contains multiple flows: for the B10 there will be four flows, and for the S10 eight flows.

The first thing we do is to iterate through each flow, looking to see if it is checked out. To do this we examine the *Checkout Bitmap* of the Timer Cache entry. If the flow is checked out then we do nothing, since checked out flows are not allowed to have timer expirations issued. If the flow is not checked out when we examine all of the timers of that flow looking for any that have expired and been reported. Such timers will have a value of all ones[51]. If any such timers exist then this flow has already been reported as having an expired timer and there is nothing extra to do. If no timers have value all ones then we check if the flow has any timer with value all ones minus one. Such timers have expired but either not yet been reported, or were previously reported but unsuccessful in issuing the CRTIMER command to the LUC. If any such timers are found then we use the *Process Expired Timer* subroutine of section 3.6.6 to process them.

[51] How can a flow not be checked out and yet have a timer that has been successfully reported? This can occur in between the time when the first expired timer event was sent to the Dispatcher, and the time when the LUC receives a lookup command for that flow. After the LUC receives the lookup command the flow will be checked out, and so no timer events are issued.

After we have examined the timers of each flow in the cache line, we must decide what to do with the cache line itself. If the *Checkout Bitmap* of this Timer Cache line is zero, then none of the flows are checked out. In that case we simply write the Timer Cache line back to its appropriate position in the Timer Table and issue an *Unlock and Remove* command to the Timer Cache. If one or more of the flows are checked out, then the TCU simply unlocks the cache line, making it available to any other functional unit.

The final task of this flow chart is to adjust the TCU variables. First we increment the *CurrentTmrIdx*, which is the index into the Timer Table. If, after incrementing this variable, it points beyond the Timer Table then we must wrap back round to the start. We do this by setting *CurrentTmrIdx* to zero, incrementing the modulo ten counter M10CNT, and then initializing the modulo eight counter M8CNT to the appropriate value as indicated in Table 24. We then continue processing at the *Start TCU* flow chart of Figure 37.

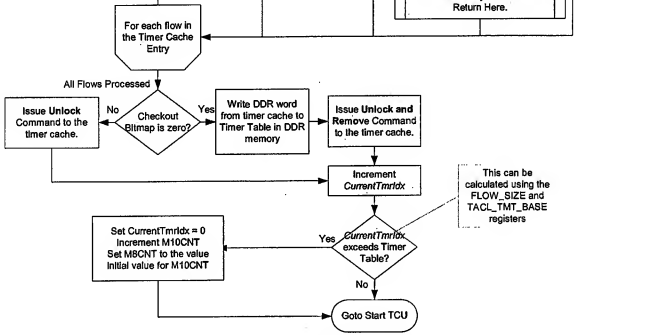

Figure 38: Process Ticked Timers Flow Chart

3.6.5 Decrement Timer Subroutine

The *Decrement Timer* subroutine of Figure 39 is used to decrement a single timer value. Note that all this flow chart does is decrement the value – it does not take any actions based upon these timer values. The definition of the timer values is given in Table 7 on page 33.

The first check we make is whether this timer is disabled. A disabled timer has value zero. If it is disabled then we do not decrement it and we simply return to the calling flow chart.

If the timer has value all ones then this timer has expired, and a CRTIMER command was successfully issued to the FDC. We therefore do not change this timer value, and simply return to the calling flow chart.

If the timer has value all ones minus one then this timer has also expired, but the CRTIMER command was either not successful or has not yet been issued. In this case we simply return to the calling flow chart without making any changes.

If the timer has value 1 then it has *just* expired. In this case we set the timer value to all ones minus one to indicate that this timer has expired, but that the CRTIMER command has not yet been issued. We then return to the calling flow chart.

Finally, if all of the above checks fail then we have a timer value that is enabled and will not yet expire. We simply decrement its value and then return to the calling flow chart.

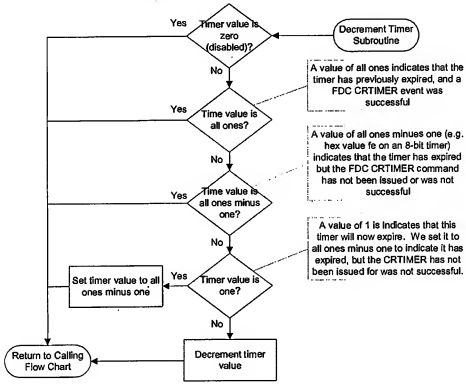

Figure 39: Decrement Timer Subroutine

3.6.6 Process Expired Timer Subroutine

The *Process Expired Timer* flow chart of Figure 40 is used when a timer of a flow that has not successfully been reported has expired. The purpose of this flow chart is to issue the correct command to the FDC and to update the timer value accordingly.

First we must determine what the flow key is. The flow key is required so that we can issue the CRTIMER command to the FDC. Note that by virtue of the value of the *CurrentTmrIdx*, and since we know the exact flow that is being processed in the Timer Cache, we know what the Socket ID of the expired timer is. Using

this value we can read the first 32 bytes[52] of the flow state for this particular flow, the format of which is defined in Figure 8 on page 17. This portion of the flow state contains the flow key.

A CRTIMER command is then issued to the Flow Director CAM, and the TCU waits for the response. If the FDC response indicates an error condition then we set a bit in the STATUS register and then return to the caller. If the FDC response indicates that this FDC entry is now in the *TIMER* state, then the CRTIMER command was successful. If the FDC indicates any other response then the CRTIMER command was not successful and must be retried at a later date. The CRTIMER command may not be successful due to a variety of reasons, e.g. the FDC may be full, or the FDC entry may be in a state that does not allow timer expirations to be issued. The reader is directed to the *Flow Director CAM* HLD for further details.

If the CRTIMER command was successful then we issue a timer event to the Dispatcher using information from FDC response. Note that the timer event for the Dispatcher requires both the Flow Key *and* the Socket ID. We then set the timer value to all ones to indicate that this timer has had a successful CRTIMER response, and to prevent any future timers for this flow from issuing another CRTIMER command.

If the CRTIMER command was not successful then the timer value is left at all ones minus one. This will have the effect of causing the TCU to retry this timer expiration the next time it processes this flow, which will be in 200ms. Note that it is possible for this timer expiration to repeatedly get unsuccessful CRTIMER responses, in which case the TCU will keep on trying every 200ms. This may lead to a timer expiration being indefinitely delayed.

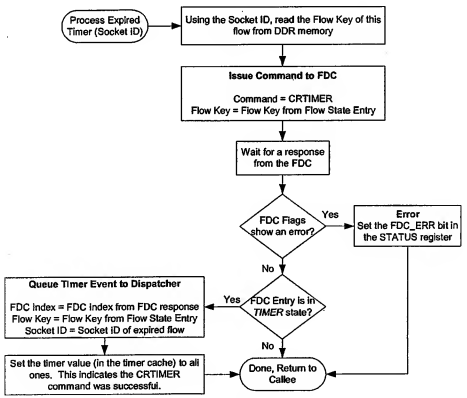

Figure 40: Process Expired Timer Subroutine

[52] For the S10 this will be the first 64-bytes.

3.6.7 Timer Coherence

In this section we show how timer events are not missed, and how the interaction with the FDC ensures that CRTIMER commands are only issued when required. This is done using with the aid of Figure 41 that shows the various transitions that a single flow goes through as its timers expire.

A flow starts in the bottom left with the *No Timers Expired* state. Each time the TCU ticks this flows timers they are decremented, until eventually one of them expires and is given the value all ones minus one. The TCU then issues a CRTIMER command: either it will be successful, in which case we enter the *Timer Expired, FDC Entry Has Been Created* state, or it fails and we enter the *Timer Expired, FDC Entry Not Created* state.

If the CRTIMER was not successful, then the TCU did not change the value of the timer from all ones minus one. This means that each time the TCU ticks this flow, it will keep on trying to issue the CRTIMER command until eventually it succeeds and enters the *Timer Expired, FDC Entry Has Been Create* state[53]. As it enters this state it issues the timer expired event to the Dispatcher.

If the CRTIMER was successful the first time round, then the TCU changed the timer value to all ones and issues the timer expired event to the Dispatcher. It then entered the *Timer Expired FDC Entry Has Been Created* state.

A flow remains in the *Timer Expired, FDC Entry Has Been Created* state until a lookup command is received for this flow. The Dispatcher will issue such a lookup command after it gets the timer expiration event. Until such a lookup command is received, the TCU ticks for this flow will not cause a CRTIMER to be issued since one of the timer values is all ones (see Figure 40).

When the LUC receives a lookup command for this flow it enters the *Timer Expired, Flow Checked Out* state. In this state TCU ticks are will not cause CRTIMER commands to be issued since the *Checkout Bitmap* for this flow will have value one. At some point the LUC will receive an update or teardown command for this flow. We know that the Processor Core must service all timers that had value all ones: either it reset or disabled those timers. However, there may still be some timers that have value all ones minus one[54]. If there are such timers then upon and update we transitions to the *Timer Expired, FDC Entry Not Created* state, in which case the next TCU tick will cause a CRTIMER to be issued. If there are not timers with value all ones minus one, or if the flow is torn down, then we enter the *No Timers Expired* state.

[53] There is no limit on how long the flow can remain in the *Timer Expired, FDC Entry Not Created* state. That depends on how busy the FDC is, how many events are being received for that flow etc. The TCU does its best effort to issue the expired timer.

[54] It is recommended that the Processor Cores also service timers with value all ones minus one. This prevents extra LUC, Dispatcher and FDC transactions from occurring.

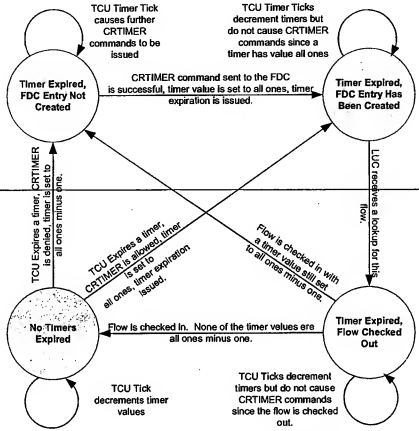

Figure 41: Timer Transitions For A Single Flow

From the above we can see that no timers are missed, and that CRTIMER commands are repeatedly issued to the FDC until one succeeds. Note also that after the first CRTIMER command is successful, no future CRTIMER commands are issued until the flow is checked back in.

3.7 LUC Counters and Statistics

The LUC counters and statistics can be split into four groups.

3.7.1 Group 1, Debug

These counters are useful for LUC debug. They count the different types of commands as they enter the LUC. See sections 4.8.28 through 4.8.31 for more details.

3.7.2 Group 2, LUC Activity

These count actions that the LUC takes in response to commands. This includes such things as:

- Entries created in response to a lookup.
- Entries found in response to a lookup.
- Entries that were not created and not found in response to a lookup.
- Updates completed.
- Teardowns completed.
- Commands dropped due to lack of resources.

See sections 4.8.32 through 4.8.37 for more details.

3.7.3 Group 3, DDR Bandwidth Use

These track DDR resources that have been used. This includes such things as:

- DDR reads/writes associated with command execution.
- TCU reads/writes.
- Housekeeping reads/writes.

See sections 4.8.38 through 4.8.43 for more details.

3.7.4 Group 4, Denial of Service

The final group of counters is useful for monitoring LUC performance and utilisation. These are grouped under a *Denial of Service* section since they might be used to detect and pre-empty denial of service attacks, e.g. a low number of available flow states may trigger SYN cookies to be activated. These counters include such things as:

- The total number of flow/listen key mismatches when searching a hash list.
- The maximum hash list length that has been encountered.
- The number of available flow state entries.
- The number of available overflow hash table entries.
- The number of available listen state entries.

See sections 4.8.55 and 4.8.57 for more details.

3.8 Expected Performance

In the following sections we examine the performance requirements for the LUC. Note that these performance requirements are not all required at the same time. For example, the connections per second metric, bulk data metric and the timer metric are all disjoint: we use the connections per second metric or the bulk data metric or the timer metric but not all at the same time.

Throughout these sections we use two models of connections. The first is the pull model, where the host CPU is informed of data being available and pulls it from the ACP. The second is a push model, where the host CPU is sent data as soon as it arrives, and it is not placed in the receive buffer.

3.8.1 LUC Performance Based on Connections Per Second Metric

This metric is based on a number of TCP connections being established and torn down with very little data transfer in between, e.g. lots of HTTP connections. Table 25 defines the performance requirements. These figures come from the *S10 vs. B10 Performance* document, available on the Intranet[55].

Host API Mode	S10	B10
Push Model Connections Per Second.	500K	200K
Pull Model Connections Per Second	385K	140K

Table 25: Connections Per Second Performance Requirements

[55] The *S10 vs. B10 Performance* document defines the maximum performance figures. These were then slightly lowered for the marketing numbers.

3.8.1.1 Pull Model

Table 26 defines the commands that the LUC will execute *per connection* for a connections per second metric using the Push model API. Using this table, Table 25, and the figures from the *B10 vs. S10 Performance* document, it would be possible to determine the load on the LUC and the DDR memory. It should be noted that for the B10 part there will also be a load on the DDR bus due to SMC transactions: the *B10 vs. S10 Performance* document has the details.

LUC Command Pair	Description
LFLKC, USID	First SYN arrives from the client. Causes a SYN+ACK to be transmitted. Reads a listen state, plus read and write a hash entry, writes a flow state.
LFK, USID	ACK of SYN from client. Three-way handshake is complete. Reads a flow state, a hash entry and writes a flow state.
LFK, USID	HTTP GET from client. Causes a data indication to be sent to the host CPU. Reads a flow state, a hash entry and writes a flow state.
LSID, USID	Application read of the HTTP GET from the host CPU. Causes a data response to be sent to the host CPU. Reads a flow state, a hash entry and writes a flow state.
LSID, USID	Application data write of a HTTP response from the host CPU. Causes a TCP segment to be transmitted. Reads a flow state, a hash entry and writes a flow state.
LFK, USID	FIN from client. Causes a Close Indication to be sent to the host CPU, and an ACK to be sent to the client. Reads a flow state, a hash entry and writes a flow state.
LSID, USID	Close Request from the host CPU. Causes a FIN to be sent to the client. Reads a flow state, a hash entry and writes a flow state.
LFK, TDFK	ACK of FIN from client. Causes the flow to be torn down. Read a flow state, plus read and write a hash entry.

Table 26: LUC Performance Based on the Connections Per Second Metric [Pull Model]

3.8.1.2 Push Model

Table 27 defines the commands that the LUC will execute *per connection* for a connections per second metric using the Pull model API. Using this table, Table 25, and the figures from the *B10 vs. S10 Performance* document, it would be possible to determine the load on the LUC and the DDR memory. It should be noted that for the B10 part there will also be a load on the DDR bus due to SMC transactions: the *B10 vs. S10 Performance* document has the details.

LUC Command Pair	Description
LFLKC, USID	First SYN arrives from the client. Causes a SYN+ACK to be transmitted. Reads a listen state, plus read and write a hash entry, writes a flow state.
LFK, USID	ACK of SYN from client. Three-way handshake is complete. Reads a flow state, a hash entry and writes a flow state.
LFK, USID	HTTP GET from client. The data is sent (pushed) directly to the host CPU via a data response message. Reads a flow state, a hash entry and writes a flow state.
LSID, USID	Application data write of a HTTP response from the host CPU. Causes a TCP segment to be transmitted. Reads a flow state, a hash entry and writes a flow state.
LFK, USID	FIN from client. Causes a Close Indication to be sent to the host CPU, and an FIN+ACK to be sent to the client. Reads a flow state, a hash entry and writes a flow state.
LFK, TDFK	ACK of FIN from client. Causes the flow to be torn down. Read a flow state, plus read and write a hash entry.

Table 27: LUC Performance Based on the Connections Per Second Metric [Push Model]

3.8.2 LUC Performance Based on Bulk Data Transfer Metric

The Bulk Data Transfer metric is used to determine the rate at which data can be transferred once a connection has been established. Table 28 defines the performance requirements for the B10 and S10 parts using both the Push and Pull API models. These figures come from the *S10 vs. B10 Performance* document, available on the Intranet[56].

Bulk Data Mode	S10 [Gbps Full Duplex]	B10 [Gbps Full Duplex]
Push Model Bulk Data	10.0Gbps	4.5Gbps
Pull Model Bulk Data	8.0Gbps	2.5Gbps

Table 28: Bulk Data Performance Requirements

3.8.2.1 Pull Model

Table 29 illustrates the LUC commands that will be received during the pull model bulk data transfer metric. This metric consists of full sized Ethernet packets being transmitted both to and from the host.

LUC Command Pair	Description
LFK, USID	Maximum sized TCP segment arrives from the network. Causes a Data Indication to be sent to the host CPU. Reads a flow state, a hash entry and writes a flow state.
LSID, USID	Application data read from the host CPU. Reads a flow state, a hash entry and writes a flow state.
LSID, USID	Application data write from the host CPU. Causes a maximum sized TCP segment to be sent to the network. Reads a flow state, a hash entry and writes a flow state.

Table 29: LUC Performance Based on the Bulk Data Transfer Metric [Pull Model]

When all of the events in Table 29 are processed, a maximum sized Ethernet packet would have been received from the network, and a maximum sized packet would have been transmitted. Therefore, using 1518 as the packet size in bytes, and allowing for a 20 byte Ethernet overhead, this produces (1518+20) * 8 bits, full duplex, on the network. Using this, and Table 28, it is possible to determine how many iterations of Table 29 are required to achieve the desired performance levels. It should be noted that for the B10 part there will also be a load on the DDR bus due to SMC transactions: the *B10 vs. S10 Performance* document has the details.

3.8.2.2 Push Model

Table 30 defines the LUC commands that will be received during the push model bulk data transfer metric. These commands are the same as the pull model, the difference being that the host does not issue an application read.

LUC Command Pair	Description
LFK, USID	Maximum sized TCP segment arrives from the network. Causes a Data Response to be sent to the host CPU. Reads a flow state, a hash entry and writes a flow state.
LSID, USID	Application data write from the host CPU. Causes a maximum sized TCP segment to be sent to the network. Reads a flow state, a hash entry and writes a flow state.

Table 30: LUC Performance Based on the Bulk Data Transfer Metric [Push Model]

[56] The *S10 vs. B10 Performance* document defines the maximum performance figures. These were then slightly lowered for the marketing numbers.

Using the same math as section 3.8.2.1 above, it is possible to determine how many of the transactions defined in Table 30 will occur per second.

3.8.3 LUC Performance Based on the Timer Metric

The final metric for the LUC is how many timer expirations it can handle per second. For both the B10 and the S10 part the performance requirement is to service 500,000 timer expirations per 200ms. Table 31 defines the steps and LUC commands that are involved in servicing a timer.

Using Table 31 it is possible to determine the LUC engine and DDR memory load. Note: it is assumed that during this metric no network traffic is present.

LUC Command Pair	Description
CRTIMER [To FDC]	A timer has expired. A CRTIMER is issued to the FDC. We assume that the FDC response indicates a success, and that the LUC forwards a Timer Event to the Dispatcher.
LSID, USID	The Dispatcher processes the Timer Event, and issues a lookup with Socket ID command. The Processor Cores process and timer expiration and then write the flow state back to the LUC. Reads a flow state, a hash entry and writes a flow state.

Table 31: LUC Performance Based on the Timer Metric

4 Interfaces

4.1 Dispatcher LUC Bus

For information on the Dispatcher to LUC bus, including the format of the LUC commands, the reader is referred to the *Dispatcher HLD*.

4.2 LUC Dispatcher Bus

4.2.1 External Signals

For information on the LUC to Dispatcher bus, including the format of the LUC timer events, the reader is referred to the *Dispatcher HLD*.

4.2.2 LUC Timer Queue

The LUC Timer Queue can hold 16 Timer Events. Each Timer Event is 145-bits[57], therefore requiring a queue of size 16 x 145-bits. If this queue becomes full then the LTQ_FULL bit of the STATUS register is set.

4.3 FDC LUC Dispatcher Bus

For information on the FDC to LUC bus (via the Dispatcher) the reader is referred to the *Dispatcher HLD*. For information on the format of the messages on this bus the reader is referred to the *Flow Director CAM HLD*.

4.4 LUC FDC Dispatcher Bus

For information on the LUC to FDC bus (via the Dispatcher) the reader if referred to the *Dispatcher HLD*. For information on the format of the messages on this bus the reader is referred to the *Flow Director CAM HLD*.

[57] This does not include the *Spare* bits. The reason is that the LUC does not queue these spare bits: it only adds them when it places the data on the LUC Dispatcher Bus.

4.5 Message Bus, Type A Transactions (From LUC)

Message Bus Type A transactions are used to send workspaces from the LUC to a Processor Core, and to validate or invalidate existing workspaces.

4.5.1 Workspace Format

Figure 42 illustrates the format of the workspace that is presented to the protocol core. In this document we only define the portions of that format which the LUC must parse, leaving the rest of the workspace format to the discretion of the software.

The Protocol Cluster provides a valid bit for Figure 42 that should be read before processing each event. It could be the case that during protocol core processing the LUC may set the workspace to invalid: this is only a temporary condition, and it does not invalidate any changes that have already been made by the protocol core. This condition occurs when the LUC executes the RMFIDX FDC command: the LUC clears the valid bit before the RMFIDX command but it does not know that there are new outstanding events until it gets the response from the FDC, at which point it will mark the workspace as valid again.

The timer values of Figure 42 were valid at the point when the flow was first checked out from the LUC. The LUC does not modify these timer values while the flow is in the protocol core's workspace, but it does modify them in the timer cache.

Table 32 provides descriptions of the other fields in the workspace.

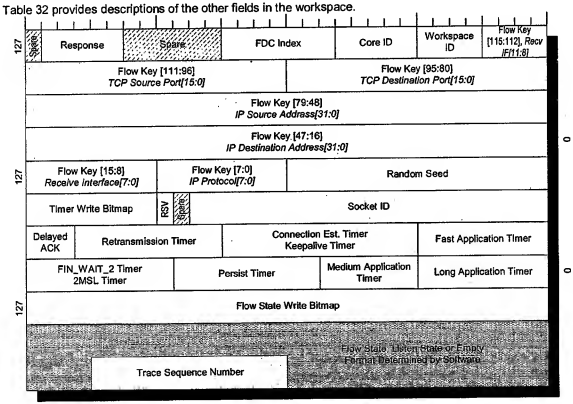

Figure 42: Workspace Format

Field	Description
Response	Used to indication various reasons as to why this workspace exists. See Table 33 for further details. The Processor Core may change this value. Note that the LUC will never set this field to zero.
FDC Index	Flow Director CAM Index. The LUC sets field using the LUC request command. This is only used for when the protocol core writes the workspace back to the LUC. The Processor Core is not required to set this field. The Processor Core must not change this value.
Core ID	This is used by the LUC to place the workspace in the correct buffer when the LUC receives it from the Message Bus. The Processor Core must not change this value.
Workspace ID	This is the Workspace ID that was presented to the LUC in the lookup command. The LUC uses this field to determine where to write the workspace in the LUC Workspace In Buffer when it receives it back from the processor core.
Flow Key	This is the 116-bit key that describes a flow. The Processor Core must not change this value. For IP protocols such as TCP and UDP, the text in *italics* shows where the various fields are located. The LUC requires the fields to be in these positions so that it can check the flow key against Termination Access Control Lists.
Timer Write Bitmap	Setting bit i of this bitmap indicates that timer ID i should be written by the LUC. Writing a timer value cancels any pending expiration. See Table 34 for the mapping from Timer to Timer ID.
RSV	This is the Random Seed Valid bit. If this bit is set then the Random Seed value is a valid random number, otherwise the Random Seed value should be ignored.
Socket ID	If the ACP is configured to support N_{FLOWS} flows, then the Socket ID will range from 0 to N_{FLOWS}-1. It is guaranteed to be unique for each flow.
Timer Fields	These are the timer values at the point when the LUC wrote the workspace. See section 2.6 for details.
Random Seed	This is a 16-bit random value that the LUC provides to the protocol core. This field is only valid if the RSV bit is set to one. If the RSV bit is zero then the Random Seed field should be ignored.
Flow State Write Bitmap	The processor cores can use this to save flow state memory bandwidth when not all of the protocol state has changed. Note that for some workspace sizes, not all of the bits in this bitmap are required. See sections 2.3.5.3 and 2.3.5.4 for information on how to interpret this field. As indicated in sections 2.3.5.3 and 2.3.5.4, when a flow is created the first block **must** be written back. This is because the LUC uses this block to store the Flow Key In, as illustrated by Figure 8. The LUC *never* initialises this field, i.e. on the first checkout it is set to an un-initialised value. The Processor Core must write a good value before it checks it in.
Trace Sequence Number	The Trace Sequence Number field is a 12-bit value that the software inserts into the command for tracing purposes only. It is not used by the LUC.

Table 32: Workspace Field Descriptions

4.5.1.1 Workspace Response Codes

Table 33 defines the response values that can appear in the response field of Figure 42.

Response Name	Value (Binary)	Flow / Listen State Valid	Description	LUC Commands that can return this response	Valid Processor Core Responses
INVALID	00000	N/A	This is never used by the LUC	None	N/A
Not found	00001	No	A lookup was performed but the flow key could not be found, and one was not created.	LFK, LFLK, LFLKC	RELS
No space	00010	No	A lookup with create LUC command was executed, but there was no space left in the hash, flow state or listen state table.	LFKC, LLKC, LFLKC	RELS
Flow found	00011	Yes Flow State	A flow key or Socket ID lookup was performed and the attached flow state was found	LFK, LFKC, LFLK, LFLKC, LSID	USID, TDFK
Listen found	00100	Yes Listen State	A flow key lookup was performed but no match was found. A listen key lookup was then performed and the attached listen state was found, but no flow was created (LFLK). An Alternative is that a regular listen lookup was performed, and the listen key was found (LLKC).	LFLK, LLKC	ULID, TDLK
Flow created	00101	No	A flow key lookup was performed but the flow key could not be found. A new flow was created.	LFKC	USID, TDFK
Listen created	00110	No	A listen key lookup was performed but the listen key could not be found. A new listen entry was created.	LLKC	ULID, TDLK
Listen found, flow created	00111	Yes Listen State	A flow key lookup was performed but the flow key was not found. A listen key lookup was then performed, and a listen entry was found. A new flow entry was then created.	LFLKC	USID, TDFK
Hash too long	01000	No	The creation of a hash entry would have made the hash list too long.	LFKC, LLKC, LFLKC	RELS
TACL Denied	01001	No	A flow key was not found and a TACL lookup was performed. That TACL lookup resulted in a DENY.	LFKC, LFLKC, LLKC	RELS

Table 33: Workspace Response Values

Note that a *Response* value of zero **will never** be used by the LUC. The Processor Cores can use this fact to determine when a workspace has been written by the LUC:

1. If the *Response* field contains a non-zero value then the LUC has just written this workspace. Before issuing the write back, the Processor Core should write this field to zero.
2. If the *Response* field contains zero, then we must have already seen this workspace due to step.

The Processor Cores can use this scheme to implement a crude timer ticking mechanism while a flow is checked out[58].

4.5.1.2 Timer Ids

Table 34 defines the mapping from timer ID to actual timer. The timer ID is used in the Timer Write Bitmap field of Figure 42.

[58] They would do this by noting the time when a workspace first arrives. Each time they process that workspace they then know how much time has elapsed since they *first* processed it.

Timer ID	Timer
0	Delayed ACK
1	Retransmission Timer
2	Connection Est. Timer or Keepalive Timer
3	Fast Application Timer
4	FIN_WAIT_2 Timer or 2MSL Timer
5	Persist Timer
6	Medium Application Timer
7	Long Application Timer

Table 34: Timer ID to Timer Mapping

4.5.2 Addressing

The LUC uses Type A message bus transactions to send to a specific Core ID using a specific Workspace ID. With reference to the Data Formats section of the *Message Bus High Level Design*, the LUC forms an address for Type A transactions as follows:

1. The Core ID is placed in the Cluster[1:0] and Core[2:0] fields of the message bus address according to the format of Figure 2.
2. The WS/E bit of the message bus address is set to 1, which means this is a workspace.
3. The Queue Entry field of the message bus address is set to the Workspace ID.
4. For sending out new workspaces, the WS/V field of the message bus address is set to 2'b00, which means *Workspace valid command from LUC disabled.* To simply clear a workspace valid bit the LUC would use value 2'b10 (*Workspace valid clear command from LUC*). To set a workspace valid bit without writing a workspace, the LUC would use 2'b11 (*Workspace valid set from LUC*)[59].

4.5.3 Backpressure

Type A transactions are sent from the LUC. Once the Message Bus has been arbitrated for, the source can never be held back.

4.6 Message Bus, Type B Transactions (To the LUC)

Message Bus Type B transactions are used by the Protocol Cluster to send workspaces to the LUC.

4.6.1 Workspace Format

The workspace format is exactly the same as that defined in section 4.5.1.

4.6.2 Addressing

The LUC only receives Type B Message Bus transactions, so it does not create an address.

4.6.3 Backpressure

Once started, Message Bus Type B transactions cannot be stopped. There is no mechanism for the LUC to apply backpressure. This is not required since Message Bus Type B transaction go directly into the LUC Workspace In Buffer as described in section 3.4.

4.7 MMC Bus

The read is directed to the *ManageMent and Control HLD* for full details on the MMC Bus.

[59] Note that when clearing or setting a workspace valid bit, the single 128-bit word that is present with the Message Bus transaction is ignored by the Protocol Cluster, i.e. it is not written to the workspace.

4.8 Configuration Registers

4.8.1 LUC Register Access

Since the LUC is in a different clock domain than the MMC, microprocessor access to its registers is slightly different from other blocks.

4.8.1.1 Read Access

Read access to the LUC is the same as any other block on the MMC, i.e. the register is read, the LUC responds with an ACK, and the microprocessor gets valid data.

4.8.1.2 Write Access

To allow for bursting, the MMC does not implement acknowledges for microprocessor writes. For this reason, it is possible that a series of back-to-back writes can arrive at the LUC. Since the LUC is in a different clock domain, it may not be able to register all these writes and may therefore miss one. Write access to the LUC must therefore follow a strict set of rules. Failure to follow these rules could lead the LUC missing microprocessor writes[60]. The rules are:

1. Only one microprocessor may have a write outstanding with the LUC. This requires that either the software guarantee that only one microprocessor will ever issue writes to the LUC, or that if two microprocessors could write then they must use a semaphore register on the MMC to serialise access.
2. After issuing a register write, the microprocessor must wait until the write has completed. It does this by repeatedly reading the WRITE_BUSY register (explained later in section 4.8.3) waiting for a "completed last write" bit to go low[61]. Reading registers does not require a poll of the WRITE_BUSY bit. If an illegal access is attempted then the MMC_ERR bit of the STATUS register is set.

The above rules must be followed for every single write access to any LUC register.

4.8.1.3 Dynamic Registers

All LUC registers are static except for:

1. The CONTROL register (section 4.8.5).
2. The DEF_AD, TACL_LFKC_EN, TACL_LFLKC_EN and TACL_LLKC_EN bits of the CONFIG register (section 4.8.6).

This means that once the LUC is initialised and enabled, only the above registers (and bits) can be modified. All other registers and bits must not be modified.

[60] The microprocessor will not be aware when the LUC does miss a write.
[61] The LUC will respond to a read of this register immediately. It does not use the read as a way of holding off the microprocessor until the previous write is complete.

4.8.2 Register Map

Offset (Hex)	Mode	Register Name	Description
0000	Read Only	WRITE_BUSY	Write Busy register. Used to synchronise between the management CPU and the LUC.
0001	Read Only	STATUS	Status information, e.g. error notification bits.
0002	Read/Write	CONTROL	Control information, e.g. reset bit.
0003	Read/Write	CONFIG	Configuration information.
0004	Read/Write	DDR_MODE	DDR mode register.
0005	Read/Write	DDR_EXT_MODE	DDR extended mode register.
0006	Read/Write	LUC_PARAMS	LUC parameters register.
0007	Read/Write	TCU_CTRL	Timer Control Unit control register.
0008	Read/Write	FLOW_CNT	Defines the maximum number of flows.
0009	Read/Write	OVFLOW_CNT	Defines the number of overflow hash table entries.
000A	Read/Write	AFST_BASE	Available Flow State Base register.
000B	Read/Write	AOHT_ALST_BASE	Available Overflow and Listen Base register.
000C	Read/Write	LISTEN_CNT	Defines the number of listen entries.
000D	Read/Write	TACL_TMT_BASE	Termination Access Control List and Timer Base register.
000E	Read/Write	HT_OHT_BASE	Hash Table and Overflow Hash Table Base register.
000F	Read/Write	FST_LST_BASE	Flow and Listen Table Base register.
0010	Read/Write	LISTEN_MASK1	Listen mask register for bits 31 through 0.
0011	Read/Write	LISTEN_MASK2	Listen mask register for bits 63 through 32.
0012	Read/Write	LISTEN_MASK3	Listen mask register for bits 95 through 64.
0013	Read/Write	LISTEN_MASK4	Listen mask register for bits 115 through 96.
0014	Read/Write	HASH_PARAMS	Hash Function Parameters register.
0015	Read/Write	DDR_ADDR	DDR address register for CPU access.
0016	Read/Write	HASH_MAX	Defines limits for the maximum hash list length.
0017	Read/Write	DEBUG	Debug register.
0018-0027	Read/Write	DDR_DATA	DDR data registers for CPU access.
0028	Read Only	LKCMD_CNT	Lookup command count.
0029	Read Only	UPCMD_CNT	Update command count.
002A	Read Only	TDCMD_CNT	Teardown command count.
002B	Read Only	RELSCMD_CNT	Release command count.
002C	Read Only	ENTRY_CRT_CNT	Count of entries created in the flow state table.
002D	Read Only	ENTRY_FND_CNT	Count of entries found in the flow state table.
002E	Read Only	ENTRY_NCNF_CNT	Count of entries when LUC could not create an entry.
002F	Read Only	UPDATE_CNT	Count of the number of updates of the flow state table.
0030	Read Only	TEARDOWN_CNT	Number of successful teardowns executed by the LUC.
0031	Read Only	DENIED_CNT	Number of commands denied due to tables overflow.
0032	Read Only	PENGMEMRDCT	Number of bursts read from the memory by the LUC command engines.
0033	Read Only	PENGMEMWRCT	Number of bursts written to the memory by the LUC command engines.
0034	Read Only	TCUMEMRDCT	Number of bursts read from the memory by the TCU.
0035	Read Only	TCUMEMWRCT	Number of bursts written to the memory by the TCU.
0036	Read Only	KEEPMEMRDCT	Number of bursts read from the memory by the resource keeper.
0037	Read Only	KEEPMEMWRCT	Number of bursts written to memory by the resource keeper.
0038	Read Only	FLOWHLTRVCT	Flow key hash list traverse count.
0039	Read Only	LSTNHLTRVCT	Listen key hash list traverse count.
003A	Read Only	FLWHTMAXTRVCT_A	Maximum hash list items searched (flow key).
003B	Read Only	FLWHTMAXTRVCT_B	Maximum hash list items searched (flow key).
003C	Read Only	FLWHTMAXTRVCT_C	Maximum hash list items searched (flow key).

Offset (Hex)	Mode	Register Name	Description
003D	Read Only	FLWHTMAXTRVCT_D	Maximum hash list items searched (flow key).
003E	Read Only	LSTNHTMAXTRVCT_A	Maximum hash list items searched (listen key).
003F	Read Only	LSTNHTMAXTRVCT_B	Maximum hash list items searched (listen key).
0040	Read Only	LSTNHTMAXTRVCT_C	Maximum hash list items searched (listen key).
0041	Read Only	LSTNHTMAXTRVCT_D	Maximum hash list items searched (listen key).
0042	Read Only	SECECCERROR_CNT	Count of correctable DRAM errors.
0043	Read Only	FSIDAVAIL_CNT	Count of available entries in the flow state table.
0044	Read Only	FOVFAVAIL_CNT	Count of available entries in the overflow hash table.
0045	Read Only	FLIDAVAIL_CNT	Count of available entries in the listen state table.
0046	Read/Write	EXT_CONFIG	Extended configuration information.
0047	Read/Write	PC_FLW_WR_MASK	Protocol Flow Write Mask.
0048	Read/Write	IC_FLW_WR_MASK	Interface Flow Write Mask.
0049	Read/Write	LIS_WR_MASK	Listen Write Mask.
004A	Read Only	RELS_CNT	Number of successful releases executed by the LUC.
004B-4F	N/A	SPARE	N/A
0050-0057	Read/Write	LWIB_PARS	LUC Workspace In Buffer Parameters.

Table 35: LUC Register Map

4.8.3 Write Busy (WRITE_BUSY) Register [0000H]

Bits	Name	Description
0-30	N/A	N/A
31	WBUSY	If the LUC is currently executing a write operation, then a read of this bit will return a 1. Since the LUC runs in a different clock domain than the MMC and since writes executed by the MMC are not acknowledged, the management CPU is not allowed to issue a new access until the WBUSY bit is cleared.

Table 36: Write Busy Register Bit Definitions

4.8.4 Status (STATUS) Register [0001H]

Bits	Name	Description
0	DUMP_IDL	This is the Dumper Idle flag. This is a flow through flag, i.e. it is not cleared on read or write, but is a flow through representation of some status.
1	FEED_IDL	This is the Feeder Idle flag. This is a flow through flag, i.e. it is not cleared on read or write, but is a flow through representation of some status.
2	TD_ERROR	This is the teardown error bit. If set then an engine attempted to teardown a non-existent flow. Cleared on read.
3	LTQ_FULL	This is the LUC Timer Queue full bit. This is a sticky bit that is set if the LUC Timer Queue ever becomes full. Cleared on read.
4-9	N/A	N/A
10	TIME_OVFL	Timer Overflow. This indicates that the TCU was unable to service timers due to memory and FDC congestion. This will result in a slip of the timer interval. Cleared on read.
11	RFRSH_ERR	Refresh Error. Indicates that the memory controller could not keep up with the outstanding refresh cycles due to memory access congestion. This should never happen in normal circumstances. Cleared on read.
12	ADDR_ERR	MMC Address Decode Error. This indicates an address decode error during an attempted out of range MMC CPU access. Cleared on read. This status bit **does not** track invalid addresses that are placed in the DDR_DATA register; it tracks invalid MMC accesses only.
13	RESP_ERR	Response Error. This indicates that an unknown command code appeared at the LUC output. This is an integrity check that should never trigger. Cleared on read.

Bits	Name	Description
14	GEN_ERR	A general error occurred. An invalid command code or Core ID was sent to the LUC from the Dispatcher, or a USID command was received but the READY bitmap was not set. This is an integrity check that should never trigger. Cleared on read.
15	MBUS_ERR	Message Bus Error. This indicates an MBUS to LUC write data framing error. This is an integrity check that should never trigger. Cleared on read.
16	DCMD_ERR	Dispatcher Command Error. This indicates that a Dispatcher to LUC write data framing error occurred. This is an integrity check that should never trigger. Cleared on read.
17	MMC_ERR	MMC Error. This is triggered when the MMC attempts either a read or a write of the LUC while the LUC is still executing the previous MMC write. Cleared on read.
18	SEC_ECC_ERR	DDR DRAM read data single bit error that was corrected by the ECC logic. Cleared on read.
19	DET_ECC_ERR	DDR DRAM read data multi-bit error that was not corrected by the ECC logic. Cleared on read.
20	TCACHE_ERR	A timer cache error occurred. See Figure 18.
21	FDC_ERR	An FDC error occurred during an RMFIDX command. See Figure 35 and Figure 36.
22	INIT_IN_PROG	Initialisation In Progress flag. This indicates that a DDR DRAM initialisation is in progress. This flag goes high when the CPU invokes either a clear or fill increment action and goes low upon completion. This is a flow through flag, i.e. it is not cleared on read or write, but is a flow through representation of some status.
23	PROG_IN_PROG	Programming In Progress flag. This indicates that a DDR DRAM mode programming is in progress. The flag goes high when the CPU initialises the programming operation and drops low upon operation completion.
24	DDR_UN_PROG	DDR Un-programmed flag. This is the default mode of the DDR, and indicates that the DDRM DRAM has not yet been programmed. All DDR operations are blocked until the DDR DRAM is programmed. Upon completion of the DDR DRAM mode programming the flag goes low. The CPU may bring the memory to the default mode by clearing the ACT bit in the CONTROL_1 register. The flag goes high when the CPU initialises the programming operation and drops low upon operation completion.
25-31	N/A	N/A

Table 37: Status Register Bit Definitions

4.8.5 Control (CONTROL) Register [0002H]

Bits	Name	Description
0	KEEP_EN	This is the Keeper Enable bit. Setting this to value 1 activates the feeder, dumper and fetcher blocks of the keeper. The dumper is the block that takes overflowing pointers from the Free Block Management FIFO specified in section 2.7 and deposits them in the appropriate queues in DDR memory. It is also responsible for clearing and initialising DDR DRAM, and also executes DDR DRAM writes in the name of the MMC. The feeder takes pointers from DDR DRAM and feeds them into the Free Block Management FIFO. Fetchers are state machines, one per FIFO, that scan the engines and in the case an engine has an empty pointer register the fetcher loads it immediately with the available pointer.
1	ACT	This is the Activate bit. This instructs the LUC to initialise flow state memory. The LUC state machine loads the corresponding mode registers and goes through the DDR DRAM's initialisation sequence. Once set high this bit should remain high throughout the operation of the LUC.
2	INIT	This is the Initialise bit. Setting this to 1 instructs the dumper to write the defined amount of words to the DDR memory. If the FILLI flag (see below) is low then the dumper clears the area starting at the value of the AFST_BASE register, and continuing for FLOW_CNT.CNT DDR bursts. See the FILLI description for what happens when the FILLI bit is high. This is used to clear areas and initialise available resource tables.
3	FILLI	This is the Fill Increment bit. This bit causes all of the available tables to be filled with incrementing 32-bit values, with each Available Table starting at value zero. The following registers must be configured before this bit is set: 1. Number of flows (FLOW_CNT. CNT).

Bits	Name	Description
		2. Number of overflow hash entries (OVFLOW_CNT. CNT). 3. Number of listen entries (LISTEN_CNT. CNT). 4. Base of the Available Flow State Table (AFST_BASE). 5. Base of the Available Overflow Hash Table (AOHT_ALST_BASE.AOHT_BASE). 6. Base of the Available Listen State Table (AOHT_ALST_BASE.ALST_BASE). This bit must be used in conjunction with the INIT bit defined above.
4	RFRESH	This is the Refresh Enable bit. When high this activates the DDR memories automatic refresh mechanism running at the refresh cycle frequency.
5	TTICK	This is the Timer Tick Enable bit. This enables the Timer Control Unit (TCU) by enabling the timer ticker generator.
6-25	MASK	This is the interrupt enable mask for bits 2-21 of the status register.
26	N/A	N/A
27	GTM_CLR	This is the Global Timer Clear bit. Setting this bit to value 1 and then value 0 causes the Protocol Clusters global timer to be cleared. See section 2.10 for further details.
28	SBINIT	This is the Scoreboard Init command bit. The Check In and Check Out buffer scoreboards reset command. This is used as part of the soft reset procedure.
29	SRESET	This is the Soft Reset bit.
30	EN_RECV	Enables the Dispatcher Interface.
31	EN_RBG	Enables the sampling of the Random Bit Generator. See section 2.9.

Table 38: Control Register Bit Definitions

4.8.6 Configuration (CONFIG) Register [0003H]

Bits	Name	Description
0-19	GTM_CYCLE	This is the cycle of the global timer. This is expressed in terms of the Protocol Cluster's clock. The minimum value of register is 10, therefore providing a minimum cycle time of 37.594ns for a 266MHz Protocol Cluster clock. See section 2.10 for further details.
20	DEF_AD	This is the default Allow/Deny bit for TACL. It is used if no match is found in the TACL table.
21	TACL_LFKC_EN	Enable TACL lookups for the LFKC command.
22	TACL_LFLKC_EN	Enable TACL lookups for the LFLKC command.
23	TACL_LLKC_EN	Enable TACL lookups for the LLKC command.
24-28	STLINE_CNT	This is the *Streamline Count* value, and is used by the DDR memory controller. To reduce the latency in switching rows, the DDR memory controller will attempt to read from the same row of memory chips if possible. The value of STLINE_CNT is such that the DDR controller is not allowed to read from the same row more than STLINE_CNT times without switching to another row (if a request for another row exists). Setting this value correctly ensures that DDR requests are not starved out indefinitely, while still allowing the DDR controller to hide the latency of switching rows as much as possible. The default value of this register should allow 512 bytes to be read without interruption, i.e. STLINE_CNT should have the value 512 / 32 = 16.
29-31	N/A	N/A

Table 39: Configuration Register Bit Definitions

4.8.7 DDR Mode (DDR_MODE) Register [0004H]

Bits	Name	Description
0-15	MODE	The MODE value is loaded into the DDR DRAMS during initialisation as a first mode value. Must reset the DRAM's DLL. See the DDR DRAM data sheet.
16-31	MODE2	The MODE2 value is loaded into the DDR DRAMS during initialisation as the second mode value. Must enable the DRAM's DLL. See the DDR DRAM data sheet.

Table 40: DDR Mode Register Bit Definitions

4.8.8 DDR Extended Mode (DDR_EXT_MODE) Register [0005H]

Bits	Name	Description
0-15	EXT_MODE	The EXT_MODE value is loaded into the DDR DRAMS during initialisation. See the DDR DRAM data sheet.
16-31	N/A	N/A

Table 41: DDR Extended Mode Register Bit Definitions

4.8.9 LUC Parameters (LUC_PARAMS) Register [0006H]

Bits	Name	Description
0	DDR_REGIST	This bit must be set to 1 only when using registered DIMMs. For non-registered DIMMS and discrete DDR DRAM chips this value must be zero.
1-2	DDR_TRCADJ	This compensates for slow memories. The DDR DRAM TRC parameters is determined by the following equation: TRC = (16 + TRCADJ) + T_{ck}
3	DDR_AFMODE	Determines the DDR DRAM address-folding mode: row and column address generation. See the figure in section A.1 for more details.
4-6	DDR_CSGEN	Determines how the DDR DRAMS chip selects are generated. The DDR_AFMODE and DDR_CSGEN values are a function of the memory type (organisation and size). See the figure in section A.1 for more details.
7	DDR_DROW	Indicates the use of DIMMs with two rows of chips per DIMM. This affects chip select generation. See the figure in section A.1 for more details.
8-10	FLOW_SIZE	Encoded value of the size of a flow state entry. This size includes the 2 x 128-bit workspace header. The encoding is: 000: 128 bytes 001: 256 bytes 010: 512 bytes 011: 1K bytes 1xx: 2K bytes
11-12	LISTEN_SIZE	Encoded value of the size of a listen state entry. This size includes the 2 x 128-bit workspace header. The encoding is: 00: 128 bytes 01: 256 bytes 10: 512 bytes 11: 1K bytes
13-19	FLW_PC_SIZE	The size, in 128-bit words, of the Protocol Cores portion of the split flow workspace. For exclusive flow state splitting (section 2.3.5.3) the value of (FLW_PC_SIZE + 2) when expressed in bytes must be a multiple of 64-bytes. For flow state splitting with sharing (section 2.3.5.4) the value of FLW_PC_SIZE when expressed in bytes must be a multiple of 64-bytes.
20-25	LIS_PC_SIZE	The size, in 128-bit words, of the Protocol Cores portion of the split listen workspace. For exclusive flow state splitting (section 2.3.5.3) the value of (LIS_PC_SIZE + 2) when expressed in bytes must be a multiple of 64-bytes. For flow state splitting with sharing (section 2.3.5.4) the value of LIS_PC_SIZE when expressed in bytes must be a multiple of 64-bytes.
26	WS_SPLIT_EN	This is the Workspace Split Enable bit. Setting this bit to value 1 enables workspace splitting.
27	ECC_EN	Setting this bit enables the SECDED operation. This bit must not be set in a configuration that does not use ECC extended memories.
28-29	DDR_CHIP_WD	This is the DDR chip width. It defines the width of memory chips (either individual or on DIMMs) used to implement the flow state memory. These two bits are interpreted as follows: 00: 4 chips 01: 8 or 16 chips 10: 32 chips 11: Invalid.

Bits	Name	Description
30	DDR_R2WTURN	This controls the DDR DRAM read to write DQ and DQS bus turnaround time. If DDR_R2WTURN has value zero then the turnaround time is: 1.5 * Tclk266 – 0.75ns – TpClk – TpDQ Where: Tclk266 is the DDR clock TpClk is the propagation time of Clk and Clk_n from Pegasus to the DDR. TpDQ is the propagation time of DQ/DQS from DDR to Pegasus. If this turnaround value becomes negative then the DDR_R2WTURN value should be set to 1. This causes an extra 2 * Tclk266 to be added to the above equation, i.e. 3.5 Tclk266 clocks are used instead of 1.5. The default value of DDR_R2WTURN is zero.
31	N/A	N/A

Table 42: LUC Parameters Register Bit Definitions

4.8.10 Timer Control Unit Control (TCU_CTRL) Register [0007H]

The TCU_CTRL register controls various parameters for the timer control unit (TCU). The INTERVAL value defines the period between successive timer ticks. Logically, for each timer tick the timers of eight flows are processed[82].

The value set for INTERVAL depends upon the maximum number of flows configured into the LUC, and the resolution of the timers. If there are 2,097,152 (2M) flows then it will take 2M / 8 ticks to tick all flows. Assuming that the required per flow resolution is 200ms, then the period between servicing each entry is 200ms / (2M / 8). This value should be converted into the number of T_{clk266} ticks and written to the INTERVAL portion of the TCU_CTRL register[83], i.e. INTERVAL = T_{clk266} * 200ms / (2M / 8).

Bits	Name	Description
0-17	INTERVAL	Timer ticker interval. This defines the period for timer ticks. Measure in T_{clk266} ticks. INTERVAL = T_{clk266} * 200ms / (N_{FLOWS} / 8).
18-19	N/A	N/A
20-31	DDR_RCYC	This is the DDR DRAMs refresh cycle. The value should be such that DDR_RCYC * 4 * T_{CLK} is less than or equal to 7.8us.

Table 43: Timer Control Unit Control Register Bit Definitions

[82] This is true for the S10 part. For the B10 part, since each Timer Table entry only contains four flows, the INTERVAL needs to be divided by two since we need to visit the Timer Table twice as often. This division (shift) is performed by the LUC, allowing the same register definition and value to be used for B10 and S10.

[83] If T_{clk266} = 266MHz then the value of INTERVAL for 2M flows at a 200ms timer resolution should be 202.94, so 203 is the value used. This rounding will cause the actual resolution to be 200.057ms rather than 200ms.

4.8.11 Flow Count (FLOW_CNT) Register [0008H]

Bits	Name	Description
0-29	CNT	This determines the maximum number of flows. The number of flows, F, is given by F = CNT * 8. Note that only bits [19:0] are used for calculating F, and that if bit 19 is set then bits 18-0 are suppressed. This gives a maximum number of flows of 4 million. The size of a flow state is determined by the FLOW_SIZE field of the LUC_PARAMS register (see section 4.8.9). This register is re-used for the DDR clear operation. In this case the full 30-bits of CNT are used, and it represents a DDR burst size (32 bytes for B10, 64 bytes for S10). If bit 29 is set then bits 28-0 are suppressed. This gives a maximum addressable range of 16GB for the B10 and 32GB for the S10.
30-31	N/A	N/A

Table 44: Flow Count Register Bit Definitions

4.8.12 Overflow Hash Count(OVFLOW_CNT) Register [0009H]

Bits	Name	Description
0-19	CNT	This determines the number of overflow hash table entries. The number of overflow hash table entries, N, is given by N = CNT * 8. If bit 19 of CNT is set, then bits 18-0 are suppressed..
20-31	N/A	N/A

Table 45: Overflow Hash Count Register Bit Definitions

4.8.13 Available Flow State Table Base (AFST_BASE) Register [000AH]

Bits	Name	Description
0-17	AFST_BASE	Base address of the Available Flow State Table = AFST_BASE * 2^{16}. This is a byte address. Note that when calculating the base of the Available Flow State Table, only bits 0-15 of AFST_BASE are used. This value is also used in conjunction with the clear operation of the LUC. In this case it is used as the base address for the clear operation.
19-31	N/A	N/A

Table 46: Available Flow State Base Register Bit Definitions

4.8.14 Available Overflow and Listen Base (AOHT_ALST_BASE) Register [000BH]

Bits	Name	Description
0-15	AOHT_BASE	Base address of the Available Overflow Hash Table = AOHT_BASE * 2^{16}. This is a byte address.
16-31	ALST_BASE	Base address of the Available Listen State Table = ALST_BASE * 2^{16}. This is a byte address.

Table 47: Available Overflow and Listen Base Register Bit Definitions

4.8.15 Listen Count (LISTEN_CNT) Register [000CH]

Bits	Name	Description
0-16	CNT	This determines the maximum number of listen entries. The number of listen entries, L, is given by L = CNT * 8. Note that if bit 16 is set then bits 15-0 are suppressed. The size of a listen state is determined by the LISTEN_SIZE field of the LUC_ PARAMS register (see section 4.8.9).
17-31	N/A	N/A

Table 48: Listen Count Register Bit Definitions

4.8.16 Termination Access Control List and Timer Base (TACL_TMT_BASE) Register [000DH]

Bits	Name	Description
0-15	TACL_BASE	Base address of the Termination Access Control List Table = TACL_BASE * 2^{16}. This is a byte address.
16-31	TMT_BASE	Base address of the Timer Table = TMT_BASE * 2^{16}. This is a byte address.

Table 49: Termination Access Control LIst and Timer Base Register Bit Definitions

4.8.17 Hash and Overflow Hash Base (HT_OHT_BASE) Register [000EH]

Bits	Name	Description
0-15	HT_BASE	Base address of the Hash Table = HT_BASE * 2^{16}. This is a byte address.
16-31	OHT_BASE	Base address of the Overflow Hash Table = OHT_BASE * 2^{16}. This is a byte address.

Table 50: Hash and Overflow Hash Base Register Bit Definitions

4.8.18 Flow and Listen State Base (FST_LST_BASE) Register [000FH]

Bits	Name	Description
0-15	FST_BASE	Base address of the Flow State Table = FST_BASE * 2^{16}. This is a byte address.
16-31	LST_BASE	Base address of the Listen State Table = LST_BASE * 2^{16}. This is a byte address.

Table 51: Flow and Listen State Base Register Bit Definitions

4.8.19 Listen Mask 1 (LISTEN_MASK1) Register [0010H]

Bits	Name	Description
0-31	LMASK_31_0	This represents the listen mask bits 31 through 0. If bit i is set in this mask, then that bit is also included in the listen key.

Table 52: Listen Mask 1 Register Bit Definitions

4.8.20 Listen Mask 2 (LISTEN_MASK2) Register [0011H]

Bits	Name	Description
0-31	LMASK_63_32	This represents the listen mask bits 63 through 32. If bit i is set in this mask, then that bit is also included in the listen key.

Table 53: Listen Mask 2 Register Bit Definitions

4.8.21 Listen Mask 3 (LISTEN_MASK3) Register [0012H]

Bits	Name	Description
0-31	LMASK_95_64	This represents the listen mask bits 95 through 64. If bit i is set in this mask, then that bit is also included in the listen key.

Table 54: Listen Mask 1 Register Bit Definitions

4.8.22 Listen Mask 4 (LISTEN_MASK4) Register [0013H]

Bits	Name	Description
0-19	LMASK_115_96	This represents the listen mask bits 115 through 96. If bit i is set in this mask, then that bit is also included in the listen key.
20-31	N/A	N/A

Table 55: Listen Mask 4 Register Bit Definitions

4.8.23 Hash Function Parameters (HASH_PARAMS) Register [0014H]

Bits	Name	Description
0-4	SHIFT_1	Shifting value: see of S_1 from section 2.3.2.
5-9	SHIFT_2	Shifting value: see of S_2 from section 2.3.2.
10-12	K_FOLD	Number of bits to fold. See the variable K from section 2.3.2. This effectively sets the size of the hash table as follows: K_FOLD = 0: 24-bit hash value, 16M hash table entries. K_FOLD = 1: 23-bit hash value, 8M hash table entries. K_FOLD = 2: 22-bit hash value, 4M hash table entries. K_FOLD = 3: 21-bit hash value, 2M hash table entries. K_FOLD = 4: 20-bit hash value, 1M hash table entries. K_FOLD = 5: 19-bit hash value, 512K hash table entries. K_FOLD = 6: 18-bit hash value, 256K hash table entries. K_FOLD = 7: 17-bit hash value, 128K hash table entries. K_FOLD = 8: Invalid.
13-15	N/A	N/A
16-27	HL_DENY_LIM	This is the hash lock deny limit value. If a LUC Engine is denied more than HL_DENY_LIM times then it is put in the high priority pool. Recommended value is 1024. If the HL_DENY_LIM is set to zero then the deny limit feature is disabled and no engines will enter the high priority pool. See section 3.2.3.3 for further details.
28-31	N/A	N/A

Table 56: Hash Function Parameters Register Bit Definitions

4.8.24 DDR Address (DDR_ADDR) Register [0015H]

The LUC provides a debug path to the flow state memory via the DDR_ADDR and DDR_DATA registers.

4.8.24.1 S10 Access

To read a 64-byte value the CPU writes the address to the DDR_ADDR register and then reads from DDR_DATA[0]. To write a value to the flow state memory the CPU first writes the address to DDR_ADDR, and then writes the data to the DDR_DATA registers. The write to DDR_DATA[15] should be done last since this triggers the actually write to DDR memory.

4.8.24.2 B10 Access

This is exactly the same as for S10 access except that a 32-byte word is used, and that a write to DDR_DATA[7] triggers the actual write to DDR memory.

Bits	Name	Description
0-28	CPU_ADDR	Address to be read/written by CPU. For the S10 this is the address for a 64-byte word. For the B10 it is a 32-byte word.
24-31	N/A	N/A

Table 57: DDR Address Register Bit Definitions

4.8.25 Hash Maximums (HASH_MAX) Register [0016H]

Bits	Name	Description
0-15	MAX_FKLEN	When creating a hash entry for a flow key, the hash entry will not be created if the hash list length would become greater than MAX_FKLEN. This does not count the initial hash entry that is in the hash table. If MAX_FKLEN = 0 then no overflow hash entries will be used.
16-31	MAX_LKLEN	When creating a hash entry for a listen key, the hash entry will not be created if the hash list length would become greater than MAX_LKLEN. This does not count the initial hash entry that is in the hash table. If MAX_LKLEN = 0 then no overflow hash entries will be used.

Table 58: Hash Maximums Register Bit Definitions

4.8.26 Debug (DEBUG) Register [0017H]

Bits	Name	Description
0-5	MUX_SEL	Debug multiplexer select. This selects what debug information is exported via the debug bus. See the LUC implementation documentation for details.
6	BYPASS	When set to 1 this connects the debug input to the debug output. When low the debug information of the LUC is inserted into the debug chain.
7-14	N/A	N/A
15	DDR_DO_FLIPEN	This is the DDR flip enable bit. If set then bits are flipped according to the DDR flip mask. Default value is zero. See section 2.11 for further details.
16-31	DDR_DO_FLIP	This field is used to write the internal DDR flip mask. See section 2.11 for further details.

Table 59: Debug Register Bit Definitions

4.8.27 DDR Data (DDR_DATA) Registers [0018H – 0027H]

This is a block of sixteen 32-bit wide registers. The first register (at the lowest LUC register address) is defined as DDR_DATA[0]. The last register of this block (at the highest LUC register address) is DDR_DATA[15]. Note that DDR_DATA[0] contains the 32-bit word at the lowest DDR address. For example, if during a DDR read DDR_DATA pointed to a TACL entry of Figure 11 then DDR_DATA[0] would contain the 32-bit word with the IP Protocol, and DDR_DATA[7] would contain the spare word. See the description of the DDR_ADDR register for further details (section 4.8.24).

Bits	Name	Description
0-31	CPU_DATA	Data to be read/written by the CPU.

Table 60: DDR Data Register Bit Definitions

4.8.28 Lookup Command Count (LKCMD_CNT) Register [0028H]

Bits	Name	Description
0-31	LKCMD_CNT	Counter of lookup commands that have been fed to the LUC. Cleared on read.

Table 61: Lookup Command Count Register Bit Definitions

4.8.29 Update Command Count (UPCMD_CNT) Register [0029H]

Bits	Name	Description
0-31	UPCMD_CNT	Counter of update commands that have been fed to the LUC. Cleared on read.

Table 62: Update Command Count Register Bit Definitions

4.8.30 Teardown Command Count (TDCMD_CNT) Register [002AH]

Bits	Name	Description
0-31	TDCMD_CNT	Counter of teardown commands that have been fed to the LUC. Cleared on read.

Table 63: Teardown Command Count Bit Definitions

4.8.31 Release Command Count (RELSCMD_CNT) Register [002BH]

Bits	Name	Description
0-31	RELSCMD_CNT	Counter of release commands that have been fed to the LUC. Cleared on read.

Table 64: Release Command Count Register Bit Definitions

4.8.32 Entries Created Count (ENTRY_CRT_CNT) Register [002CH]

Bits	Name	Description
0-31	CRT_CNT	Count of new entries created in the flow state tables as a response to "Lookup and Create" commands. Cleared on read.

Table 65: Entries Created Count Register Bit Definitions

4.8.33 Entries Found Count (ENTRY_FND_CNT) Register [002DH]

Bits	Name	Description
0-31	FND_CNT	Count of entries found in the flow state table as a response to lookup commands. Cleared on read.

Table 66: Entries Found Count Register Bit Definitions

4.8.34 Entries Not Created and Not Found Count (ENTRY_NCNF_CNT) Register [002EH]

Bits	Name	Description
0-31	NCNF_CNT	Count of instances when the LUC was unable to find and create an entry as a result of a lookup command. Cleared on read.

Table 67: Entries Not Created and Not Found Count Register Bit Definitions

4.8.35 Update Count (UPDATE_CNT) Register [002FH]

Bits	Name	Description
0-31	UPDATE_CNT	Number of successful updates of the flow state table. Cleared on read.

Table 68: Update Count Register Bit Definitions

4.8.36 Teardown Count (TEARDOWN_CNT) Register [0030H]

Bits	Name	Description
0-31	TD_CNT	Number of successful teardowns executed by the LUC. Cleared on read.

Table 69: Teardown Count Register Bit Definitions

4.8.37 Denied Count (DENIED_CNT) Register [0031H]

Bits	Name	Description
0-31	DENIED_CNT	Number of commands denied due to tables overflow. Cleared on read.

Table 70: Denied Count Register Bit Definitions

4.8.38 LUC Engine Memory Read Count (PENGMEMRDCT) Register [0032H]

Bits	Name	Description
0-31	RD_CNT	Number of DDR bursts read from the memory by the LUC command engines. Cleared on read.

Table 71: LUC Engine Memory Read Count Register Bit Definitions

4.8.39 LUC Engine Memory Write Count (PENGMEMWRCT) Register [0033H]

Bits	Name	Description
0-31	WR_CNT	Number of DDR bursts written to the memory by the LUC command engines. Cleared on read.

Table 72: LUC Engine Memory Write Count Register Bit Definitions

4.8.40 TCU Engine Memory Read Count (TCUMEMRDCT) Register [0034H]

Bits	Name	Description
0-31	RD_CNT	Number of DDR bursts read from the memory by the Timer Control Unit engines. Cleared on read.

Table 73: TCU Engine Memory Read Count Register Bit Definitions

4.8.41 TCU Engine Memory Write Count (TCUMEMWRCT) Register [0035H]

Bits	Name	Description
0-31	WR_CNT	Number of DDR bursts written to the memory by the Timer Control Unit engines. Cleared on read.

Table 74: TCU Engine Memory Read Count Register Bit Definitions

4.8.42 Keeper Memory Read Count (KEEPMEMRDCT) Register [0036H]

Bits	Name	Description
0-31	RD_CNT	Number of DDR bursts read from the memory by the resource keeper. Cleared on read.

Table 75: Keeper Memory Read Count Register Bit Definitions

4.8.43 Keeper Memory Write Count (KEEPMEMWRCT) Register [0037H]

Bits	Name	Description
0-31	WR_CNT	Number of DDR bursts written to the memory by the resource keeper. Cleared on read.

Table 76: Keeper Memory Write Count Register Bit Definitions

4.8.44 Flow Hash List Traverse Count (FLOWHLTRVCT) Register [0038H]

Bits	Name	Description
0-31	TRV_CNT	Flow key hash list traverse count. During lookup operations this counts the misses when searching through the hash list with the flow key. This is a running total. Cleared on read.

Table 77: Flow Hash List Traverse Count Register Bit Definitions

4.8.45 Listen Hash List Traverse Count (LSTNHLTRVCT) Register [0039H]

Bits	Name	Description
0-31	TRV_CNT	Listen key hash list traverse count. During lookup operations this counts the misses when searching through the hash list with the listen key. This is a running total. Cleared on read.

Table 78: Listen Hash List Traverse Count Register Bit Definitions

4.8.46 Flow Hash Max Traverse Count (FLWHTMAXTRVCT_A) Register [003AH]

Bits	Name	Description
0-15	LUCE0_MAX	This is a count of the maximum length hash list chain that LUC Engine #0 encountered during a flow key search. Cleared on read.
16-31	LUCE1_MAX	This is a count of the maximum length hash list chain that LUC Engine #1 encountered during a flow key search. Cleared on read.

Table 79: Flow Hash Max Traverse Count Register Bit Definitions

4.8.47 Flow Hash Max Traverse Count (FLWHTMAXTRVCT_B) Register [003BH]

Bits	Name	Description
0-15	LUCE2_MAX	This is a count of the maximum length hash list chain that LUC Engine #2 encountered during a flow key search. Cleared on read.
16-31	LUCE3_MAX	This is a count of the maximum length hash list chain that LUC Engine #3 encountered during a flow key search. Cleared on read.

Table 80: Flow Hash Max Traverse Count Register Bit Definitions

4.8.48 Flow Hash Max Traverse Count (FLWHTMAXTRVCT_C) Register [003CH]

Bits	Name	Description
0-15	LUCE4_MAX	This is a count of the maximum length hash list chain that LUC Engine #04encountered during a flow key search. Cleared on read.
16-31	LUCE5_MAX	This is a count of the maximum length hash list chain that LUC Engine #5 encountered during a flow key search. Cleared on read.

Table 81: Flow Hash Max Traverse Count Register Bit Definitions

4.8.49 Flow Hash Max Traverse Count (FLWHTMAXTRVCT_D) Register [003DH]

Bits	Name	Description
0-15	LUCE6_MAX	This is a count of the maximum length hash list chain that LUC Engine #6 encountered during a flow key search. Cleared on read.
16-31	LUCE7_MAX	This is a count of the maximum length hash list chain that LUC Engine #7 encountered during a flow key search. Cleared on read.

Table 82: Flow Hash Max Traverse Count Register Bit Definitions

4.8.50 Listen Hash Max Traverse Count (LSTNHTMAXTRVCT_A) Register [003EH]

Bits	Name	Description
0-15	LUCE0_MAX	This is a count of the maximum length hash list chain that LUC Engine #0 encountered during a listen key search. Cleared on read.
16-31	LUCE1_MAX	This is a count of the maximum length hash list chain that LUC Engine #1 encountered during a listen key search. Cleared on read.

Table 83: Listen Hash Max Traverse Count Register Bit Definitions

4.8.51 Listen Hash Max Traverse Count (LSTNHTMAXTRVCT_B) Register [003FH]

Bits	Name	Description
0-15	LUCE2_MAX	This is a count of the maximum length hash list chain that LUC Engine #2 encountered during a listen key search. Cleared on read.
16-31	LUCE3_MAX	This is a count of the maximum length hash list chain that LUC Engine #3 encountered during a listen key search. Cleared on read.

Table 84: Listen Hash Max Traverse Count Register Bit Definitions

4.8.52 Listen Hash Max Traverse Count (LSTNHTMAXTRVCT_C) Register [0040H]

Bits	Name	Description
0-15	LUCE4_MAX	This is a count of the maximum length hash list chain that LUC Engine #4 encountered during a listen key search. Cleared on read.
16-31	LUCE5_MAX	This is a count of the maximum length hash list chain that LUC Engine #5 encountered during a listen key search. Cleared on read.

Table 85: Listen Hash Max Traverse Count Register Bit Definitions

4.8.53 Listen Hash Max Traverse Count (LSTNHTMAXTRVCT_D) Register [0041H]

Bits	Name	Description
0-15	LUCE6_MAX	This is a count of the maximum length hash list chain that LUC Engine #6 encountered during a listen key search. Cleared on read.
16-31	LUCE7_MAX	This is a count of the maximum length hash list chain that LUC Engine #7 encountered during a listen key search. Cleared on read.

Table 86: Listen Hash Max Traverse Count Register Bit Definitions

4.8.54 SEC ECC Error Count (SECECCERROR_CNT) Register [0042H]

Bits	Name	Description
0-31	ERR_CNT	Count of the number of correctable DRAM errors. Cleared on read.

Table 87: SEC ECC Error Count Register Bit Definitions

4.8.55 Free Flow State Count (FSIDAVAIL_CNT) Register [0043H]

Bits	Name	Description
0-25	FREE_CNT	This is a count of the number of flow state entries that are available. This is a flow through value, i.e. it is constantly updated and never cleared.
26-31	N/A	N/A

Table 88: Free Flow State Count Register Bit Definitions

4.8.56 Free Overflow Hash Entry Count (FOVFAVAIL_CNT) Register [0044H]

Bits	Name	Description
0-18	FREE_CNT	This is a count of the number of overflow hash entries that are available. This is a flow through value, i.e. it is constantly updated and never cleared.
19-31	N/A	N/A

Table 89: Free Overflow Hash Entry Count Register Bit Definitions

4.8.57 Free Listen State Count (FLIDAVAIL_CNT) Register [0045H]

Bits	Name	Description
0-15	FREE_CNT	This is a count of the number of listen state entries that are available. This is a flow through value, i.e. it is constantly updated and never cleared.
16-31	N/A	N/A

Table 90: Free Listen State Count Register Bit Definitions

4.8.58 Extended Configuration (EXT_CONFIG) Register [0046H]

Bits	Name	Description
0-7	PC_TMR_MASK	This is the mask that is applied to the *Timer Write Bitmap* field of the Protocol Core Workspace (Figure 42) before any of the Timer Write Bitmap bits are examined. This register allows control over what timers the Protocol Core can reset.
8-15	IC_TMR_MASK	This is the mask that is applied to the *Timer Write Bitmap* field of the Interface Core Workspace (Figure 42) before any of the Timer Write Bitmap bits are examined. This register allows control over what timers the Interface Core can reset.
16-21	FLW_SH_SIZE	This is the size, in 128-bit words, of the shared area of a flow state. The value of (FLW_SH_SIZE + 2), when expressed in bytes, must be a multiple of 64-bytes. See section 2.3.5.4 for more details.
22-26	LIS_SH_SIZE	This is the size, in 128-bit words, of the shared area of a listen state. The value of (LIS_SH_SIZE + 2), when expressed in bytes, must be a multiple of 64-bytes. See section 2.3.5.4 for more details.
27-31	N/A	N/A

Table 91: Extended Configuration Register Bit Definitions

4.8.59 Protocol Flow Write Mask (PC_FLW_WR_MASK) Register [0047H]

Bits	Name	Description
0-31	MASK	This mask is applied to the Flow State Write Bitmap field of a flow state workspace header (see Figure 42) for a Protocol Core before any blocks are written back. It allows blocks to be protected from being overwritten.

Table 92: Protocol Flow Write Mask Register Bit Definitions

4.8.60 Interface Flow Write Mask (IC_FLW_WR_MASK) Register [0048H]

Bits	Name	Description
0-31	MASK	This mask is applied to the Flow State Write Bitmap field of a flow state workspace header (see Figure 42) for an Interface Core before any blocks are written back. It allows blocks to be protected from being overwritten.

Table 93: Interface Flow Write Mask Register Bit Definitions

4.8.61 Listen Write Mask (LIS_WR_MASK) Register [0049H]

Bits	Name	Description
0-15	IC_LIS_MASK	This is the lower 16-bits of the mask that is applied to the Flow State Write Bitmap field of a listen state workspace header (see Figure 42) for an Interface Core before any blocks are written back. It allows blocks to be protected from being overwritten. The upper 16-bits of the mask are assumed to be zero since the maximum size of a listen entry is 1KB
16-31	PC_LIS_MASK	As above, but for a Protocol Core.

Table 94: Listen Write Mask Register Bit Definitions

4.8.62 Release Count (RELS_CNT) Register [004AH]

Bits	Name	Description
0-31	RL_CNT	Number of successful releases executed by the LUC. Cleared on read.

Table 95: Release Count Register Bit Definitions

4.8.63 LUC Workspace In Buffer Parameters (LWIB_PARS) Registers [0050H – 0057H]

The LUC Workspace In Buffer Parameters register blocks provides parameters for managing the LUC Workspace In Buffer. Logically there is a set of LUC Workspace In Buffer parameters for each Processor Core. We store two sets of parameters per register, so there are eight LWIB_PARS registers in total.

These registers are indexed by a Core Index value, as specified in section 2.1.2. To find the LUC Workspace In Buffer parameters for a register first select the LWIB_PARS register using Core Index[3:1], and then use Core Index[0] to select (BASE_0, WS_SHIFT_0) or (BASE_1, WS_SHIFT_1).

Bits	Name	Description
0-13	BASE_0	This is the offset to use for this specific Core Index into the LUC Workspace In Buffer. This is a 128-bit word address.
14-15	WS_SHIFT_0	This is the amount to shift the Workspace ID by before it is used to index into the LUC Workspace In Buffer. Possible values are 0, 1 or 2. 3 is not a valid value. This should match the shift parameter that is programmed into the Protocol Cluster registers.
16-29	BASE_1	Same definition as BASE_0 above.
30-31	WS_SHIFT_1	Same definition as WS_SHIFT_0 above.

Table 96: LUC Workspace In Buffer Register Bit Definitions

4.9 *Initialisation*

The following sequence should be used to initialise the LUC:

1. Bring the LUC out of reset. At this point DDR_UN_PROG bit of the STATUS register will be high.
2. Wait 200µs and then load the DDR DRAMs physical parameters:
   a. Set the DDR_REGIST, DDR_TRCADJ, DDR_AFMODE, DDR_CSGEN and DDR_DROW fields of the LUC_PARAMS register to the appropriate values.
3. Set the DDR_MODE and DDR_EXT_MODE registers accordingly. Then set the ACT bit of the CONTROL register. This bit must be kept high on all future writes of the CONTROL register. After setting the ACT bit, the LUC will perform the following actions:
   a. Activate CKE output.
   b. Execute a NOP command.
   c. Execute a PreChargeAllBanks command.
   d. Load the DDR DRAMs ExtendedModeReg register with the EXT_MODE field of the DDR_EXT_MODE register on the LUC.
   e. Load the DDR DRAMs ModeReg register with the MODE field of the DDR_MODE register on the LUC. This will reset the memory DLL.
   f. Execute a PreChargeAllBanks command.
   g. Execute two AutoRefresh commands in a row.
   h. Load the DDR DRAMs ModeReg with DDR_MODE[{14:9},{15},{7:0}] to program operating parameters without resetting the DLL.
   i. Wait 400 clock cycles to let the DLL acquire clock lock.
   j. Drive low (inactive) the PROG_IN_PROG and DDR_UN_PROG bits of the STATUS register.
4. Wait for the PROG_IN_PROG bit of the STATUS register to go low.

5. Set the DDR_RCYC field of the TCU_CTRL register to an appropriate value.

6. Enable automatic refresh by setting the RFRESH bit of the CONTROL register. This cannot be done before steps 1 through 5 since the automatic refresh state machine is disabled when the DDR_UN_PROG bit of the STATUS register is low.

7. At this point the DDR memory can be used. The software **must** initialise the Hash Table to zero. To do this:
   a. Load the base of the Hash Table into the AFST_BASE register.
   b. Load the size, in DDR bursts, into the CNT field of the FLOW_CNT register.
   c. Set INIT=1 and FILLI=0 in the CONTROL register in one write.
   d. Poll the STATUS register until INIT_IN_PROG goes low. The LUC signals completion of the memory initialisation by clearing this bit.

8. The Timer Table **must** also be cleared to zero. To do this follow the same steps as defined above for clearing the Hash Table.

9. Each Available Table must now be initialised with an incrementing 32-bit value, starting at zero. The LUC initialises all Available Tables in one go using the FILLI bit of the CONTROL register. To do this, the following sequence must be followed:
   a. Configure the AFST_BASE field of the AFST_BASE register with the base of the Available Flow State Table.
   b. Configure the AOHT_BASE field of the AOHT_ALST_BASE register with the base of the Available Overflow Hash Table.
   c. Configure the ALST_BASE field of the AOHT_ALST_BASE register with the base of the Available Listen Table.
   d. Configure the CNT field of the FLOW_CNT register with the number of flow entries.
   e. Configure the CNT field of the OVFLOW_CNT register with the number of overflow hash entries.
   f. Configure the CNT field of the LISTEN_CNT register with the number of listen entries.
   g. Set INIT=1 and FILLI=1 in the CONTROL register in one write.
   h. Poll the STATUS register until INIT_IN_PROG goes low. The LUC signals completion of the memory initialisation by clearing this bit. The Available Tables are then initialised.

10. Enable the resource keeper logic by setting KEEP_EN bit of the CONTROL register. The keeper pre-fetches pointers to the entries in the tables into the proper FIFOs. This is part of the initialisation of the Free Block Manager.

11. All other registers in the LUC that have not already been explicitly configured must now be initialised. The settings for these registers are application specific.

12. Set and then clear the SBINIT bit of the CONTROL register. The TCU interval should then be set in the INTERVAL field of the TCU_CTRL register. Setting the TTICK bit in the CONTROL register enables the TCU.

13. Enable the reception of commands from the Dispatcher and messages from the Message Bus by setting EN_RECV bit of the CONTROL register. This will activate the luc_dsp_ready output and activate the LUC input logic.

14. Done.

5 Design Rationale

5.1 Hash Entry Size

With reference to section 2.3.3, there is a considerable amount of spare space in the S10 hash entry format. The reason for this is that the S10 LUC has a minimum burst size of 64-bytes to the DDR flow state memory. To keep things simple it was decided that the hash entry size should match this burst size.

Possible Use of Spare Space	Issues
We could store extra flow state in the 46-bytes. This would be in addition to the flow state stored under the Socket ID pointer.	This complicates an update. When the Dispatcher issues a flow state update it does so with the Socket ID. This allows the LUC to directly update memory, without locking any data structures. If it were also required to update the data in the hash entry then it would need a back pointer from the flow state to the hash entry. It would also need to lock the hash list on flow state updates.
We could store another hash entry for a different hash value.	This is really saying that we could put part of the overflow hash entry table in the hash table. This would really complicate things for the LUC.
We could store another hash entry for the same hash value. It may even be possible to fit three hash entries per 64-byte word.	This complicates the hash table maintenance algorithms for the LUC. It now has to allow for two keys being in the same hash entry. However, doing this does mean that a hash list of length two could be accessed in a single shot.

Table 97: S10 Hash Entry Compression Methods

5.2 LUC Workspace In Buffer

There was much discussion to do with decreasing the size of the LUC Workspace In Buffer described in section 3.4. The argument was that the LUC does not need to buffer sixteen workspaces: using a smaller number would still keep the LUC busy and would decrease the buffer memory requirements.

One thing to note about the workspace writes from the protocol core is that the workspaces and LUC commands do not arrive in sequence, i.e. a workspace could arrive from protocol core number four, but the next LUC update command could be from protocol core number two. This reordering is due to the message bus being arbitrated amongst multiple cluster controllers.

5.2.1 LUC Workspace In CAM Possible Solution

A suggested scheme was to use eight workspace buffers that are shared between all protocol cores. There would be a single buffer full bit that would be presented to all cluster controllers. An eight-entry CAM would manage this buffer, with the key to the CAM being the FDC index. Each entry in the CAM would contain a pointer to the workspace (one of eight) and a reference count. This CAM was designed to overcome the order issue.

Workspaces received from the message bus would first be placed in a holding buffer, and the buffer full flag would be set. We would then lookup the FDC index in the CAM. If a match was found then the workspace pointed to by the CAM entry would be overwritten, the reference count would be incremented, and the holding buffer would be cleared.

If a CAM entry were not found then a CAM entry would be created with the reference count set to one. If there were space for at least one more entry in the CAM then the buffer full flag would be cleared, otherwise it would be left on. This ensures that we always have room for one entry in the CAM.

Update, teardown and discard commands from the Dispatcher would allow the LUC to decrement the reference count of the CAM entries. When a reference count reaches zero the CAM entry can be removed, possibly allowing the buffer full flag to be lowered.

While this scheme appears to work, it is obviously more complicated than a simple buffer per protocol core. This complexity could increase both design, verification and debug time. For this reason we decided to use

the simpler, but more memory intense, buffer per protocol core. However, this option is always available if we decide that trade off for space is worth the extra complexity and time.

6 Open Issues

None.

7 Summary

The preceding sections should have accurately described the operation of the LUC. Please notify the author of any discrepancies, omissions or typos.

Appendix A

A.1 DDR DRAM Address Folding

The following diagram illustrates the way the LUC folds the DDR DRAM address for various memory configurations. This is used to set the DDR_AFMODE field of the LUC_PARAMS register of section 4.8.9.